A Cooperative Caching Architecture for Improving Database Queries

Doctoral Thesis

For the Degree of
Doctor of Informatics

At the Faculty of Economics,
Business Administration And
Information Technology
of the
University of Zurich

by
Andrei Aurel Vancea
from
Cluj Napoca, CJ, Romania

Accepted on the recommendation of
Prof. Dr. Burkhard Stiller
Prof. Dr. Torsten Braun

February 2013

Abstract

Client side caching is a commonly used technique for reducing the response time of database queries. Semantic caching is a database caching approach in which results of old queries are cached and used for answering new queries. A new query will be split in a part that retrieves the portion of the result, which is available in a local cache (probe query) and a query that retrieves missing n-tuples for the database server (remainder query). This approach is especially suited for low-bandwidth environments or when database server is under heavy load. Semantic caching was successfully applied for optimizing the execution of queries on mobile clients or over lousy-coupled wide-area networks. Peer-to-Peer (P2P) networks have been applied successfully for improving the performance of cache systems. This thesis aims to use a P2P approach in order to enhance the semantic caching technique.

Therefore, The Cooperative Semantic Caching (CoopSC) approach extends the general semantic caching mechanism by using a P2P (Peer-to-peer) approach in order to enable clients to share their local semantic caches in a cooperative manner. When executing a query, the content of both the local semantic cache and entries stored in caches of other clients can be used. A new query will be split into a probe, remote probes, and a remainder query. The probe retrieves the part of the answer which is available in the local cache. Remote probes retrieve those parts of the query which are available in caches of other clients. The remainder retrieves the missing n-tuples from the server. In order to execute the query rewriting, the cache entries of all clients are indexed in a distributed data structure built on top of a P2P overlay that is formed by all clients which are interrogating a particular database server. The general cooperative semantic approach also contains a mechanism for handling operations that modify the content of the database. During

this thesis, the general approach was applied in the context of two real-life use-cases: cloud-computing environments, and NetFlow-based network monitoring solutions.

This thesis shows that a cooperative semantic caching approach is technical feasible and such an approach approach increases the performance of databases systems. Moreover, in the context of such an approach, update handling can be executed with a reasonable performance when read-intensive workloads are executed. Finally the general CoopSC approach presents economic advantages when used within cloud-computing environments and improves the performance of NefFlow-based network monitoring solutions.

Kurzfassung

Client-seitiges Caching ist eine verbreitete Technik um die Antwortzeit von Datenbankabfragen zu reduzieren. Semantisches Caching ist ein Datenbank-Caching-Ansatz bei welchem die Ergebnisse vergangener Abfragen gespeichert und für die Antworten von neuen Abfragen verwendet werden. Hierbei wird eine neue Anfrage aufgeteilt in den Teil, welcher im lokalen Cache verfügbar ist (Probe-Anfrage) und in eine Anfrage welche die fehlenden n-Tupel vom Datenbank-Server abruft (Remainder-Anfrage). Dieser Ansatz ist besonders geeignet für schmalbandige Verbindungen oder für Datenbank-Server unter großer Last. Semantisches Caching wurde erfolgreich für die Ausführungsptimierung von Anfragen auf mobilen Clients oder über lose gekoppelte Wide-Area-Networks angewendet. Peer-to-Peer (P2P) Netzwerke wurden erfolgreich für die Verbesserung von Caching-Systemen angewendet. Ziel dieser Arbeit die Verwendung eines P2P-Ansatzes um die Semantische Caching-Technik weiterzuentwickeln.

Der Cooperative Semantic Caching (CoopSC) Ansatz erweitert das herkömmliche semantische Caching um einen P2P-Ansatz. Dies ermöglicht einen kooperativen Austausch der lokalen semantischen Caches unter den Clients. Zur Ausführung einer Anfrage kann nun sowohl der lokale sematische Cache und die Caches von anderen Clienten benutzt werden. Eine neue Anfrage wird aufgeteilt in eine Probe-Anfrage, Remote-Probe-Anfrage und eine Remainder-Anfrage. Die Probe-Anfrage ruft den Teil der Anwort ab, die im lokalen Cache verfügbar ist. Die Remote-Probe-Anfrage ruft die Teile ab, welche in den Caches der anderen Clients verfügbar sind. Die Remainder-Anfrage ruft die verbleibenden n-Tupel vom Datenbank-Server ab. Um die Anfrage umzuschreiben werden die Cache-Einträge aller Clienten in einer

verteilten Datenstruktur indiziert. Dieser Datenstruktur liegt das P2P-Netzwerk der mit dem Datenbank-Server verbundenen Clienten zugrunde. Das allgemeine CoopSC beinhaltet desweioteren einen Mechanismus um Schreib-Operation am Datenbankinhalt zu ermöglichen. Im Rahmen dieser Arbeit wurder das allgemeine CoopSC im Zusammenhang mit zwei realen Anwendungsfällen betrachtet: Cloud-Computing-Umgebungen und NetFlow-basiertes Netwerk-Monitoring.

Die Arbeit zeigt die technische Machbarkeit des CoopSC und dass ein solcher Ansatz die Leistungfähigkeit von Datenbanksystemen verbessern kann. Desweiteren kann gezeigt werden, dass mit dem allgemeinen CoopSC Schreib-Operationen mit angemessener Leistung durchgeführt werden können, wenn eine hohe Last an lesenden Anfragen besteht. Zusammenfassend stellt das allgemeine CoopSC in Cloud-Computing-Anwendungen einen ökonomischen Vorteil dar und erhöht die Leistungsfähigkeit von NetFlow-basierten Netzwerk-Monitoring-Lösungen.

Contents

1 Introduction

P2P (Peer-to-peer) systems define a type of distributed network environments in which each participant node can act as a *server* or a *client* for all other nodes from the distributed environment [73]. On one hand, compared with the classic client-server infrastructure, P2P systems are usually much more scalable since all nodes from the environments can participates in answering specific requests. On the other hand, managing such a distributed system is a difficult task [83] since it lacks a centralized way of controlling the environment, which is usually present is client-server architectures.

A way of achieving scalability in database management systems is to effectively utilize resources (storage, CPU) of client machines. Client side caching is a commonly used technique for reducing the response time of database queries [18]. Semantic caching [31] is a database caching approach, in which results of old queries are cached and used for answering new queries. A new query will be split in a part that retrieves the portion of the result that is available in a local cache (*probe query*) and a query that retrieves missing n-tuples from the database server (*remainder query*). This approach is especially suited for low-bandwidth environments or when the database server is under heavy load. Semantic

caching was successfully applied for optimizing the execution of queries on mobile clients or over loosely-coupled wide-area networks [78]. Semantic caching requires more resources on clients. Storage is needed for storing cache entries. Clients' CPU usage will also increase, because they, locally, execute the probe sub-query.

In most applications, database servers are queried by multiple clients. When using the classic semantic caching approach, clients store and manage their own local caches independently. If the number of clients is high, the amount of data sent by database server and queries response times can rapidly increase even when caching is used. The performance can be further improved by allowing clients to share their entries in a cooperative way. Another limitation of existing semantic caching solutions is that they do not handle update queries. Modification performed in the database are not propagated to cache entries stored by clients.

Peer-to-peer (P2P) networks have been applied successfully for enhancing the traditional client-server caching approaches. E.g., the CoopNet [73], uses a cooperative network caching architecture for solving Web flash crowd scalability problems. These results show that a cooperative P2P-based caching approach significantly increase the performance of client-server architectures under heavy load.

Therefore, this thesis investigates the feasibility of using a P2P approach in order to improve the performance of semantic caching solutions. The result of this investigation is the Cooperative Semantic Caching (CoopSC) approach [88, 90] which extends the general semantic caching mechanism by enabling clients to share their local semantic caches in a cooperative manner. When executing a query, the content of both the local semantic cache and entries stored in caches of other clients can be used. Before execution, a query will be split into *probes*, *remote probes*, and a *remainder* sub-queries. A probe retrieves a part of the answer, which is available in the local cache. Remote probes retrieve those parts of the query which are available in caches of other clients. Remainders retrieve the missing n-tuples from the server. In order to execute the query rewriting, the cache entries of all clients will be indexed in a distributed data structure built on top of a Peer-to-peer

(P2P) overlay that is formed by all clients which are interrogating a particular database server. Additionally, CoopSC designs a suitable and efficient mechanism for handling update queries. When the content of the database is changed, modifications are reflected in the cooperative cache. Furthermore, CoopSC supports select-project queries, where the query predicate is a n-dimensional range condition, which are commonly used in many database applications.

1.1 Motivation

CoopSC decreases the response time of database queries, because servers only handle the portions of queries that can not be answered using the cooperative cache. Also, the amount of data sent by database servers can be significantly reduced. Thus, this approach is suited for the following types of resource constrained environments: (a) Database server and clients are located in a higher-bandwidth local-area environment. Clients execute a large number of queries in parallel. In this scenario, server's processing resources (CPU, disk access) are the bottle-neck of the system. Using cooperative caching decreases queries response time because it reduces servers' resources usage; (b) Database server and clients are located across the Internet in a network-constrained environment (e.g., the infrastructure of multi-national corporation). Clients execute queries that return a large amount of data (n-tuples). In this scenario, the cooperative caching approach is beneficial because is reduces the amount of data sent by database servers.

For example, NetFlow-based traffic monitoring solutions provide a good real-life use-case which can benefit from applying a cooperative database semantic caching approach. In such approaches, NetFlow data is collected from network routers, stored in database servers and accessed by analyzers, which perform different network monitoring tasks (e.g., accounting, charging, intrusion detection). Thus, this use case, which is also investigated by this thesis, provides a good motivating example of the CoopSC approach.

In the context of these two types of application, the major aim of CoopSC is the enhancement of the performance of read-intensive query workloads. Such types of workloads are frequently used in many type of applications, including decision-support systems. Select-project queries, where the predicate is a n-dimensional range condition, are commonly used when queries dimensional data (e.g., geographic information). Thus, the real-life case is considered with a high priority. Furthermore, with the emergence of cloud computing infrastructures, using a cooperative database caching approach can have economic advantages, because cloud providers usually bill data transferred between cloud environment and the outside world. Thus, the minimization of amount of data sent by database server can achieve such a cost reduction [89].

1.2 Problem Statement

The main goal of this thesis is to extend the general semantic caching mechanism by using a P2P approach for enabling clients to share their local semantic caches in a cooperative manner in order to improve system performance. When executing a query, the content of both the local semantic cache and entries stored in caches of other clients can be used. Such an approach increases the performance of databases systems and presents economic advantages when used in a cloud-computing environment. Thus, P2P technology is used in order to develop a cooperative caching solution.

Additionally, in order to allows CoopSC to be used in real-life database applications, the overall functionality needs to reflect such real-life demands. Therefore, the approach supports select-project queries, where the query predicate is a n-dimensional range condition, which are commonly used in many database applications. Moreover, this thesis designs a suitable and efficient mechanism for handling update queries. When the content of the database is changed, modifications must be reflected in the cooperative cache. The cooperative caching approach was applied in the context of cloud computing environments and

NetFlow-based traffic management application aiming to provide both performance- and economic-wise benefits.

Therefore, in what follows, the main research questions which are answered during this thesis are clearly enumerated and described.

A. **How can cooperative aspects be integrated in the classic semantic caching approach?**

In the classic semantic caching approach each client handles its own local cache independently. A cooperative approach shall allow clients to share their local caches in a cooperative matter. Mechanisms must be provided for determining which relevant remote cache entries are useful for answering a specific query. Queries must be split into parts which are retrieved from local cache, remote caches, and database server. Remote query execution mechanisms must also be provided by such a cooperative semantic database caching solution. Determining a feasible cooperative database approach is one of the main contribution of this thesis.

B. **How to handle update statements in the context of the cooperative semantic caching approach?**

When modifications are performed, entries from the cooperative cache must be invalidated. Determining a mechanism for handling update statements within such a distributed environment is an important problem this thesis must solve. Since a semantic caching approach combines different cache entries in order to answer queries, it is important that when entries are combined they belong to the same database snapshots, otherwise inconsistencies can occur.

C. **Does the cooperative caching approach improve the performance of database solutions?**

In order to show the benefits of the cooperative semantic caching approach, its performance has to be verified and compared with a non-caching scenario and also with the classic semantic caching solution. It

order to accomplish this, a prototype CoopSC system [91] is to be designed, implemented and evaluated through experimentation. The experiments shall vary multiple parameters (e.g., query size, cache size, number of clients) and measure relevant performance metrics (e.g., response time, data transferred, hit rate).

D. **How does the update handling mechanism impact the performance of the cooperative caching approach?**

In order to determine the performance of the update handling mechanism, experiments shall be performed and show how a write intensive workload does influence the overall performance of the prototype CoopSC implementation. These experiments shall vary the ratio of update statements and other relevant parameters.

E. **How can the cooperative caching approach be applied within cloud-computing database solutions and NetFlow-based traffic monitoring applications and which are the benefits?**

This thesis also aims to investigate how the cooperative semantic caching approach can be applied in order to improve two potential use cases: database solutions which run inside cloud computing environments and NetFlow-based network monitoring application. Thus, relevant architectures and deployment scenarios are to be determined.

The potential performance- and economic-wise benefits of the two use cases are to be determined by running specific experiments. The design of these experiments shall make the real-life aspects of these two use-cases clearly visible.

1.3 Thesis Contributions

This thesis develops a cooperative semantic caching approach and an architectures which improves range selection interrogations. This approach is then applied in order to improve cloud computing database applications and NetFlow-based traffic monitoring solutions. Based on

the six research questions outlined in the previous section, the main contributions of this thesis, related to the *cooperative caching approach*, the *CoopSC architecture* and *use cases* are now described.

1.3.1 The CoopSC Approach

The CoopSC approach extends the classic semantic caching approach by allowing clients to share their local cache entries in an cooperative way. For each query a *query rewriting* process is executed in order to determine which relevant entries from local cache, remote caches of database server can be used. The general cooperative caching approach also handles update statements.

A distributed data structure that permits indexing the cache entries of all clients was designed. The purpose of the distributed index is to facilitate the query rewriting process using entries stored in the distributed cache. Given a query, the distributed index returns a list of entries that intersect the query. The distributed data structure must be able to index n-dimensional range select-project queries. The distributed index runs on top of a P2P overlay formed by all clients, which are interrogating a specific database server.

The query rewriting algorithm determines parts of queries that can be answered by using the local cache, remote caches, or the database server. The query rewriting algorithm accesses the distributed index in order to determine the remote cache entries which can be used for answering parts of a specified query. This algorithm was optimized in term of speed and scalability given the fact that the system will contain a large number of cache entries.

A mechanism for handling update queries was also designed. As the result of modifications performed in the database, some entries stored in the distributed cache must be discarded. The respective update mechanism was implemented as a component running in the database environment, which follows an optimized usage of speed. Compared with existing updates solution for materialized view approaches, CoopSC had to design new update strategies for an environment where the number of

entries is large and each entry contains a comparative smaller number of tuples.

1.3.2 The CoopSC Architecture

In order to determine the potential benefits of the cooperative database caching approach, a CoopSC architecture was designed and implemented, resulting in a CoopSC prototype system. This system was then evaluated inside a real-life geographically distributed environment. The evaluation clearly shows the performance-wise improvements of the CoopSC approach.

The CoopSC system was then deployed in cloud computing infrastructures in order to determine the potential performance- and economic-wise benefits of applying the cooperative database semantic caching approach in such contexts. Experiments were performed inside commercially available cloud computing infrastructures. The results of these experiments show that the CoopSC approach provides both technical and economical benefits when used inside cloud infrastructures.

1.3.3 The NMCoopSC Architecture

The CoopSC architecture was extended in order to improve NetFlow-based networks monitoring applications. The extended architecture was named NMCoopSC (Network Management CoopSC). This architecture allows analyzers to efficiently access NetFlow data, which was collected from network routers and stored in database servers. This NMCoopSC architecture was evaluated by running experiments which use real-life NetFlow data.

1.4 Thesis Outline

The remainder of the thesis is organized as follows.

Chapter 2 gives a detailed overview of the technical background of this thesis by presenting the most relevant related work approaches and architectures. It starts by describing the most important database

caching approaches, followed by an overview of existing cooperative caching solutions.

The general CoopSC approach is then introduced and described in Chapter 3. The *query rewriting* algorithm and the *update handling mechanism* are motivated and presented. This chapter also outlines the design of the *distributed index*, which is used for indexing cache entries.

Based on the general approach, the architecture of the CoopSC prototype system is presented in Chapter 4. The main architectural components, which are running of both client and server side, are described presenting also the interactions between them.

Chapter 5 gives an overview of the design and the implementation of the CoopsC prototype system. The UML (Unified Modeling Language) diagrams of the main components are illustrated and described. The CoopSC API (Application programming interface) is also presented.

The way in which CoopSC can be applied within cloud computing environments is highlighted is Chapter 4.5. The main scenarios are identified described, and discussed.

Chapter 4.6 shows how the CoopSC approach can be applied in order to improve NetFlow-based traffic monitoring solutions, by introducing the new NMCoopSC architecture. This architecture is motivated, described and discussed.

The results of the detailed evaluation of the CoopSC approach, and NMCoopSC architecture are presented in Chapter 6. The experiments are ran using the EmanicsLab [39] distributed testing infrastructure and thus, mimic real-life distributed use cases. The cloud scenarios are also evaluated using commercially available cloud computing environments.

Chapter 7 summarizes this thesis' results, highlighting the main contributions by given clear answers to the main research questions which were asked in this chapter. Conclusions are drawn, an outlook on possible future work is given.

2
Related Work

This chapter introduces selected related work, which is highly related to database caching, materialized views, and mobility. Thus, it founds the basis for a detailed comparison of existing work with the proposed CoopSC approach and all of its functions, performance evaluations, and use cases.f

This chapter compares the CoopSC approach with other caching existing systems presenting the main difference by outlining the main research contributions of the CoopSC approach.

Figure 2.1 illustrates the positioning of CoopSC approach within the general context of other caching solutions also indicates the main research areas which will be described and compared with CoopSC in this chapter. Thus, initially, this section will present other classic database caching approaches, putting a special emphasis on semantic caching solutions, outlining the main contributions and benefits of CoopSC (c.f. Section 2.1). Since semantic caching was also applied in the context of mobile environment, this research area will be investigated (c.f. Section 2.1.4). Afterwards, the CoopSC approach will be compared with other existing general cooperative caching approach, also outlining the main differences and contributions (c.f. Section 2.2). Furthermore, the

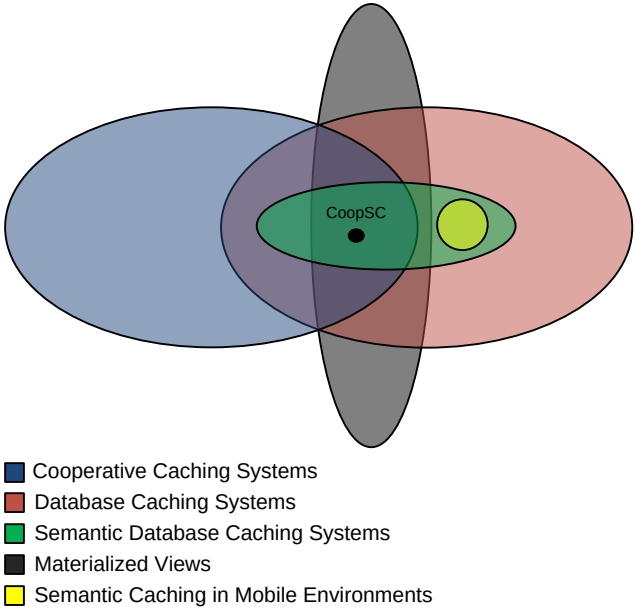

Figure 2.1: CoopSC: Related Work

CoopSC approach will be positioned in the narrowed context of other cooperative caching approaches (c.f. Section 2.3). Since the cooperative semantic caching approach has similarities with the general approach of storing materialized views, the area will also be investigated and compared with the solutions which emerged within the CoopSC project (c.f. Section 2.4). Finally, Section 2.5 presents the major concluding remarks and outlines the main contributions of CoopSC compared with the other related caching approaches.

2.1 Database Client Caching

Client side caching within database infrastructure is an important optimization technique which improves the general performance of the system by also using some of the hardware resource of clients. Since, nowa-

days, many database clients have significant hardware resource [8], using such caching techniques has beneficial effects.

Early database caching approaches aim to improve the performance of *data-shipping architecture* [8]. In such systems, it is assumed the most of the query processing is done on client side with data which is returned from database servers in advance. For example, CAD (Computer Aided Design) applications can store complex hierarchical data inside an OODBMS (Object Oriented Database Management System) [8]. Parts of these data are returned, on demand, to a client which usually perform mathematical computation locally. Caching such data can significantly improve performance.

Data-shipping systems stored cache entries either as *pages* of fixed size or as *objects* (or *tuples*). Thus, *page caching* and *object caching* emerged as the main architectural caching solutions.

Assuring the consistency of the local cache emerged as one of the main problems that has to be handled by database caching systems. [42] presents and compares the main algorithms and approaches, which were developed for handling consistency and page and object-based caching approaches. These approaches were classified into two categories: a) invalid access preventions methods [1, 18, 60, 93] and b) avoidance-based algorithms [18, 41, 61, 93, 94]. When invalid access prevention is used, transactions which access out-of-data data are not allowed to commit while avoidance-based algorithms do not allow transactions to even access stale data from their local cache.

Data-shipping architectures are not suited for all types of applications. For examples, when wanting to return a subset of data-set based on the evaluation of a predicate, a pure navigational data-shipping access will be inefficient because it would require the return of all data and the evaluation of the selection predicate locally. For such type of applications, *query-shipping architectures* are much more efficient. In such architectures, queries are sent to database servers, executed there, and results of the executions are sent back to clients. Most relational database systems are built using this type of architecture. The *semantic caching approach* [31] allows clients to cache data within query-shipping database

architectures. This approach caches locally the results of old queries, which are then used for answering new database interrogations.

This section describes and compares these approaches illustrating their key differences in Table 2.1.

2.1.1 Page Caching

When a page caching approach is used, the database server's buffer is organized into fixed-size pages which are transferred, when needed, to clients. These clients can decide to cache such pages in order to improve the overall system performance. This solution was implemented in object-oriented database management systems such as ObServer [52] and Exodus [17].

This caching approach has both drawbacks and advantages [34]. Since the database server only handles pages, most of the computational complexity of the system is moved to the client side. The server only has to handle data recovery and concurrency control. Because, nowadays, most clients have significant hardware resource, the overall performance of the system can be improved.

Due to the simplicity of the servers' design, method evaluation can only be done on client side. This can have negative consequence when executing some types of queries. For example, a sequential scan of a collection requires the transfer of all pages to client.

Another drawback is related to the implementation of the object locking mechanism. If two object are stored in the same page, locking one object will also prohibite the access to the other object since the granularity of low-level access is always on the page.

2.1.2 Object Caching

Some object oriented database systems (e.g., O_2 [12], Orion [60], GemStone [29]) use an object cache approach in order to improve performance. When using this solution, clients cache objects. When needing to access an object the local cache is first checked. If the required object is missing the request is sent to the database server.

Table 2.1: Database Caching Approaches

Approach	Architecture	Cache Entry	Systems
Page Caching	Data-shipping	Fixed-size pages	ObServer [52], Exodus [17]
Object Caching	Data-shipping	Objects (or tuples)	O_2 [12], Orion [60], GemStone [29]
Semantic Caching	Query-shipping	Semantic regions (i.e. results of old queries)	Semantic Caching [31], Predicate Caching [59], XCache [20], CoopSC [91]

The object caching approach has some advantages [34] compared with the page caching approach. For example, both the database server and clients can run methods since the object semantic exists on both sides,

The concurrency control is also simplified. Locking object is feasible since the database server controls the access to objects and knows which objects are accessed by each application.

Unfortunately, this approach has also many problems. The server design becomes much more complex because, compared with page caching solutions, it also needs to be able to run methods.

Transferring one object at a time is another potential drawback since many object can be small and thus, the general overhead of data transmission increases. A large enough local hit rate might alleviate this problem.

Since a single object can be stored on both client and server side and client updates are not always automatically propagated to server and executing a method on server side becomes problematic. The database server needs to handle all possible inconsistencies that can occur by having a single object stored in multiple places. Some approaches flush all clients' caches while other execute methods on both client and server sides.

2.1.3 Semantic Caching

The semantic caching approach which, was introduced in [31] as the basic concept, cached results of old queries and allows these results to be used for answering new queries. This paper [31] describes semantic caching concepts and compares the approach with page and tuple caching. The cache is organized into disjoint semantic regions. Each semantic region contains a set of tuples and a constraint formula, which describes the common property of the tuples. Simulations were performed for single and double attribute selection queries. These simulations show that semantic caching outperforms both tuple and page caching. [56] runs an extensive performance study of a semantic caching prototype implementation for range queries up to four attributes. Experiments were performed using the Wisconsin benchmark [14] data set, show that semantic caching decreases both the response time and the amount of data sent by database server for one and two-dimensions selection queries, while for queries with higher dimensions the decrease is only significant in regards to the amount of data sent by server. However, the classic semantic caching approach - as referred to in [31] and [56] - does not handle update queries.

The predicate caching approach, presented in [59], caches locally results of old queries together with predicates that describe cache's content. A subsequent query may be answered from local cache if it can be determine that its results are totally contained in the local cache. The approach supports select-project-join queries. Because storing and processing exact cache predicate descriptions might be computational expensive, the predicate caching solutions keeps only a conservative approximative cache description, which guarantees that data thought to be in the cache is present in it. Furthermore, compared with semantic caching [31], predicate caching approach stores cache entries that do not necessary have to be disjoint. On one hand, duplicate data could be stored in different cache entries, which might negatively influence the performance of the caching system. On the other hand, making sure the all cache entries are disjoint might be time-wise expensive, es-

pecially when more complex select-project-join queries types are supported. Predicate caching also supports update statements. Modifications are initially performed locally and only sent to database server when new local queries can not be computed locally and are sent to servers. When conflicts occur, the approach provides mechanism for canceling transactions.

XCache [20] determines a semantic caching architecture developed for XML (eXtended Markup Language) queries. The system implements algorithms for checking the query containment for XQueries and algorithms that perform query rewriting. Partial query hits are not supported. However, update queries are also not handled in [20].

[33] describes a client caching approach suited for OLAP (Online analytical processing) database management systems. Such system use highly dimensional data. The local cache is organized into fixed-size *chunks*. This approach has features from the semantic and page caching solutions [56]. Query results are broken into chunks which are stored into the local client cache. When executing new queries, the system first determines the set of chucks which intersect the given queries. The content of chunks is returned either from local cache or from the database server. Choosing the right chunks size is really important in regards to performance. A larger chunk size might returns unwanted data, since queries results are always returned in term of chunk. Having chunks which are too small increases the overhead of the system. Since update queries are uncommon in the context of OLAP system, they are not handled by this caching approach.

The DBProxy [4] caching systems aims at improving the performance of web applications by caching query results on edge applications servers. Only strict query matching is supported. The caching approach is implemented as a JDBC (Java Database Connectivity) driver that can be plugged, in a transparent ways, directly in the application server. The cache is organized into regions that can overlap. Update statements are handled by receiving an update stream from the database server.

[2] describes another application caching architecture which improves the performance of web applications, called DBCache. The ap-

Table 2.2: Database Semantic Caching Approaches

Approach	Data Model	Cache Entry	Hit Types	Updates
Semantic Caching [31]	Relational	Non-overlapping regions	Strict,Partial	No
Predicate Caching [59]	Relational	Overlapping regions	Strict,Partial	Yes
XCache [20]	XML	Overlapping regions	Strict	Yes
OLAP Chunk-Caching [33]	OLAP Relational	Chunks	Strict. Partial	No
DBProxy [4]	Relational	Overlapping regions	Strict	Yes
DBCache [4]	Relational	Tables	Strict,Partial	No
CachePortal [16]	Relational	Web pages	Strict	Yes
Semantic caching of Web queries [21]	Web-based repositories	Semantic Regions	Strict, Partial	No

proach allows caching both static and dynamic tables using a newly introduce concept of *cache table*. A cache table specified that a table in a database (*cache database*) is the cache of another table (*backened table*) from a different database (*backend database*). The architecture only supports query predicates with an equality predicate on at least one domain column and thus, range interrogations are not handled. The system also caches join interrogations. DBCache uses DB2's distributed query processing mechanism in order to decide if a particular query is to be executed using the local database cache, the remote database server or by splitting the query into subqueries which are executed on both sides.

The CachePortal [16] system enables dynamic content caching for web-based database-driven e-commerce applications. This approach caches dynamic web pages together with SQL queries which were used for generating these pages. A *sniffing* module is used for mapping web content to database interrogations. When modification are performed in the database server, an *invalidation* component removes the affected cached web pages. This component listen to the database update log in

order to determine the modifications performed in the database. Thus, this approach is similar with both the classic semantic approach, since it keeps the descriptions of queries for each cache entries, and also with web caching because the system stores, in its cache, web content.

[21] describes a semantic caching approach which improves the performance of web meta-searchers systems. The approach supports keyword-based *conjunctive* queries which are common in the context of web-based information repositories. The cache is organized into semantic regions. Each regions description contains a *regions signature* which is a binary code that allows a fast comparison among conjunctive formulas. Strict and partial cache hits are supported. Experiments show the positive performance-wise effects of applying a semantic caching approach in the context of web-queries.

Table 2.2 illustrates the key differences between these semantic caching approaches. Furthermore, all these approaches do not allow clients to share their caches in a cooperative way. Thus, only local cache entries can be used for answering queries.

2.1.4 Semantic Caching within Mobile Computing

The semantic caching approach was also successfully applied in mobile applications in order to improve the performance of location dependent data [78]. Location dependent queries select specific resource located in a area surrounding the current location (e.g., hotels within 10 km). These queries can be represented as two-dimensional range selections. Similar or same queries are often re-executed after the position of the mobile device is changed. Existing cache entries will then be used for answering new queries. Thus, a new queries is split into parts that are returned from the local caches and sub-queries which are sent to the server. A new replacement policy FAR (Furthest Away Replacement) was also developed. This policy takes into consideration the moving direction of the mobile device. Simulations show that the newly design caching approach improves the performance of location dependent queries.

[96] describes a semantic caching approach which aims at improving the performance of *mobile nearest-neighbor search* within a mobile environment. These searches retrieve stationary service objects nearest to mobile users. The approach uses an index-based on Voronoi Diagrams [9]. For each position of the mobile user, the approach defines a *semantic circle* which represents the valid scope of a answer to a specific nearest-neighbor query. Each query results contains the current nearest service, the radius of the semantic circle, the duration of validity of the current service, and the next nearest service facility. The current speed and direction of the mobile user are used in order to estimate the amount of time the current service is valid and the next nearest service.

[54] proposes a proactive mobile query architecture, based on semantic caching, which enables stored cached entries to be used by multiple type of spatial query. For example, the cache result of a simple spatial range selection can be used afterwards to answer a general *kNN* (*k* nearest neighbor) query [79]. The approach store locally a R-tree spatial index [48] of the cached query result in order to be used for efficiently answering other types of spatial interrogations.

Thus, semantic caching concepts were successfully applied in optimizing the performance of location dependent queries in the context of mobile environments. These solutions do not allow mobile client to cooperate and thus, only local cache is used for answering spatial interrogations. Using a cooperative approach in such mobile environments might not even be practical due to the battery-wise power limitation of most mobile devices.

2.2 Cooperative Caching

Cooperative caching is a common technique which improves the performance of many distributed and parallel systems.This section gives an overview of most general cooperative caching approaches.

CMP (Chip Multiprocessors) Cooperative Cache [19] improves the performance of the chip multiprocessor systems by allowing CPUs to share their private L2 caches in a cooperative way. The approach imple-

ments three cooperative policies: a) cache-to-cache transfers of clean data, b) replication-aware data replacement and c) global replacement of inactive data. The first policy (a) facilitates cache-to-cache transfers in order to reduce the number of accesses to the memory. The second policy (b) tries to avoid storing a single data block in multiple L2 private caches while policy c) allows blocks evicted from a local L2 cache to be placed in the another local cache. A central directory is used for maintaining the state of the cooperative cache

[30] studies the way in which a cooperative caching solution can improve the performance of network file systems. Clients are assumed to be located in the same LAN (Local Area Network). Local caches are organized into fixed size blocks which are stored in memory. The paper compared four cooperative caching solutions: a) direct client cooperation, b) greedy forwarding, c) central coordinated caching and d) n-chance forwarding. When using the first solution (a) clients can use the resource of other idle peers in order to cache their local data. When idle clients become active the cached block of other clients are discarded. This approach requires no change of the file server. In the *greedy forwarding* approach the server maintains the states of all clients' caches and forwards requests to clients which have the required block in their local caches. The *centrally coordinated caching* solution statically partitions the local caches into a locally managed section, which is managed individually by each client, and a globally managed section, which is managed by the server. The *n-Change forwarding* solution dynamically adjusts the fraction of each client's globally managed section based on clients' activity. These approaches are compared through simulations. The results show that solution d) outperforms the other approaches.

Cooperative caching principles were also applied in the context of mobile ad-hoc networks. For example, [95] describes an approach in which mobile nodes, which are part of an ad-hoc network, cache and share data in a cooperative way. Three approaches are presented and compared. In the first approach, named *CacheData*, nodes cache a passing-by data record locally when that record is popular. The *CachePath* approach allows client to cache paths to specific data records in order

to reduce the number of hops during the routing process. *HybridCache* combines the first two approaches.

[50] presents a mobile cooperative caching approach for push-based information systems. In such a push-based system a server repetitively broadcast various data to clients using a broadcast communication mechanism. When a mobile node wants to access a particular data item it first checks the local caches. If that required item is not present in local cache, the mobile nodes tries to find that data item in the caches of other clients. If response time of accessing the response cached entries is lower than to wait to the next broadcast the remote cache entries is transferred locally.

Other research projects aim at the provisioning of cooperative caching facilities in Web environments. For example, [73] presents CoopNet, a cooperative network architecture, where clients cooperate in order to improve the overall network performance. It is described how CoopNet is used for solving Web flash crowd scalability problems. In this approach, clients that have already downloaded Web content, start serving the content to other clients, relieving the server of this task. The redirection of requests from the server to other clients is handled by a centralized component running at the server side. Thus, this approach does not integrate the distribution aspect.

Squirrel [55] is a decentralized, P2P Web caching system. It enables Web browsers to share their local caches in a scalable matter. All Web clients are a part of a P2P overlay based on the Pastry [80] system. Each URL (Uniform Resource Locator) is associated with a node from the P2P overlay, which is called the home node. This association is done by applying a hash function on the URL and choosing the node with the closest ID to the hash value. Two approaches are implemented: Home-Store and Directory. When the Home-Store approach is used, Squirrel stores objects both at client caches and at its home node. In the Directory approach, home nodes only store the IDs of other existing nodes the have the relevant content.

Hence, cooperative caching principles were applied in optimizing many types of distributed and parallel systems, from CPU design to web

architectures. Compared with semantic caching database approaches, these systems have relatively simple data models (e.g., memory blocks, web pages). In order to answer requests, most such systems only need to access one cached entry. The CoopSC approach needs to combine multiple cache entry in order to efficiently answer relation selection queries.

2.3 Cooperative Database Semantic Caching

The Wigan system [28] caches old results of database queries in order to answer new queries and to allow for the entries cached to be shared between clients. Wigan supports only queries that can be expressed as conjunctions of single attribute range conditions. Thus, the query "select * from earthquakes where lat < 10 and long < 20" is supported, while "select * from earthquakes where lat < 10 or long < 20" is not. A cached query Q1 can be used for answering a query Q2 only, if Q2 is strictly subsumed by Q1 . In real world applications, the number of cases, in which this happens, is limited. Another drawback of this approach is that is uses a centralized tracker in order to determine, which entries cached can be used when answering a new query. A centralized approach will show in certain cases scalability and reliability problems, since the tracker represents a single point of failure. This can be avoided in a fully decentralized approach. Furthermore, Wigan does not handle update queries, too.

[63] describes a cooperative caching architecture for answering XPath queries with no predicates. Two methods of organizing the distributed cache are proposed: (a) IndexCache: each peer caches the results of its own queries; and (b) DataCache: each peer is assigned a particular part of the cache data space. The approach works with the XML data model and supports simple XPath queries that have no selection predicates. XPath queries assume a hierarchical XML structure and return a sub-tree of this structure. When answering a query, the XPath approach searches for a cache entry that strictly subsumes the given query. Thus, in consequence, partial hits are not supported. Another problem with this approach is that is does not handle update queries as well.

Table 2.3: Cooperative Semantic Caching Approaches

Approach	Data Model	Query Types	Hit Types	Resolution	Update
Wigan [28]	Relational	Simple range selections	Strict	Centralized tracker	No
XPath Index-Cache [63]	XML	XPath (no predicates)	Strict	Distributed Index	No
Dual Cache [35]	Gedeon	Non-range queries	Strict	Flooding	No
CoopSC [91]	Relational	Range select-project queries	Strict, Partial	Distributed Index	Yes
PeerOLAP [57]	Relational	OLAP	Strict, Partial	Flooding	No

The Dual Cache approach [35] is a caching service built on top of the Gedeon data management system [32]. The system performs a separation between query and object caches. It also allows cache entries of clients to be shared in a cooperative matter. The cooperation is done using a flooding approach, but the system allows new types of cache resolution to be added. In order to overcome the scalability issues of flooding, client are divided into communities. Thus, only clients that are in the same community can cooperate. Dual Cache handles non-range predicates only (e.g., lat = 20 and long = 50) and supports only strict hits between query entries. Update queries are also not handled.

The PeerOLAP [57] architecture improves the performance of OLAP relation database by allowing clients to cooperatively share their local caches. The approach uses the fixed chunk division introduced in [33]. Similar with the original approach, the system first determines chunks which are needed to answer a given query. Afterwards, the system uses a broadcast approach in order for transferring the content of the determined chunks from other clients. In order to avoid flooding the network, each TTL (time-to-live) mechanism is used. Update queries are not supported by this approach. Due to the use of a fixed chunk-based space division unneeded tuples have to be transferred when an-

swering queries because such queries can ask for only a sub-set of a particular chunk. Since the CoopSC approach uses an exact query rewriting process, it only transfers the required data.

Therefore and in summary, Table 2.3 illustrates the key differences between the cooperative semantic caching approaches investigated as related work as well as outlining already for comparing dimensions the new CoopSC approach.

2.4 Materialized Views

The cooperative semantic caching approach has similarities with the general approach of storing materialized views [47] in a distributed database environment. In both approaches, the storage unit consists of a set of n-tuples and predicates which describe the common property of these tuples. The main difference is related to the place where these entries are stored: materialized views are stored on the server site, while CoopSC stores data on the client side. An addition important difference consists in the fact that CoopSC has to store a much larger number of views which usually, contain a small number of tuples (i.e. results of old queries). These differences induce additional research challenges that must be solve by CoopSC.

Answering queries using views has been an intensively studied problem in database literature [49]. The aim is to find efficient mechanisms for using a set of previously defined materialized database views for answering database queries. Most existing approaches assume that materialized views are stored on the server side, while semantic caching keeps results of old queries on the client side. Existing approaches have two important use cases: query optimization and data integration. In query optimization, finding a query rewriting that uses existing materialized view can generate a more efficient query execution plan. Existing algorithms used for query optimization (e.g., System-R style [44], GMAP [87]), are designed to handle a small number of existing database materialized views and perform at least a linear iteration over the list of views. The number of views used by the cooper-

ative semantic approach is, usually, large because each client stores result of old queries. In the data integration use cases ([62], [46]), the goal is to find a maximally-contained rewriting using views that represent mappings of different schemas and which often does not return the complete result set. The cooperative semantic caching approach uses a single database schema and aims to always return complete result sets. In order to tackle this problem and design an efficient query rewriting algorithm, the XPath Index-Cache [63] approach indexes the XPath queries in a distributed index which resembles the Prefix Hash Tree structure. CoopSC designs an efficient distributed index that supports query rewriting of generic n-dimensional select-project database queries.

The general issue of updating materialized views in a distributed database environment remains a challenging problem [82]. There are three main types of strategies: a) update views after each change; b) defer views updating until a related query is issued; c) periodically or on-demand re-materialized views. Because the semantic caching approach stores a much larger number of views which contain a smaller fewer tuples new strategies were developed for efficiently handling update statements.

2.5 Conclusions

Existing cooperative semantic caching systems lack the support of complex query types. There are no approaches in place, which handle generic n-dimensional range selections. Another limitation of existing solutions is the way in which cache entries are used for answering a new query: existing approaches only look for an entry that strictly subsumes the query. Thus, combining multiple entries in order to answer a given query is not supported. Furthermore, most approaches do not provide a scalable way of finding which entries are suitable for answering new queries. Compared with the classic materialized views solutions, query rewriting and handling update statements in the context of cooperative semantic caching presents many additional scalability challenges which

the CoopSC approach handles. The CoopSC approach solves these challenges in a distributed environment as mentioned above.

3

The CoopSC Approach

The Cooperative Semantic Caching (CoopSC) approach, illustrated in Figure 3.1, extends the general semantic caching mechanism by enabling clients to share their local semantic caches in a cooperative manner. When executing a query, the content of both the local semantic cache and entries stored in caches of other clients can be used. A new query will be split into *probe* (2), *remote probe*, and *remainder* (3) sub-queries using a *query rewriting* process. The probe retrieves the part of the answer, which is available in the local cache (5). Remote probes retrieve those parts of the query which are available in caches of other clients (5). The remainder retrieves the missing tuples from the server (5). These sub-queries are integrated into a *query plan tree* which specifies the way in which the original query is to be executed. The final result set is cached locally in order to be used when answering new queries (6) and then returned to the user (7).

In order to execute the query rewriting, cache entries of all clients will be indexed in a distributed data structure built on top of a Peer-to-peer (P2P) overlay that is formed by all clients which are interrogating a particular database server. Additionally, CoopSC designs a suitable and

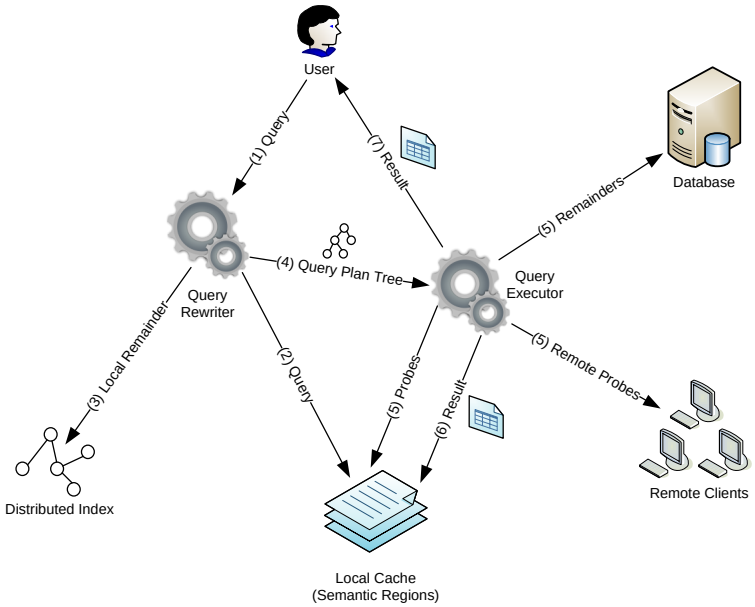

Figure 3.1: CoopSC Approach

efficient mechanism for handling update queries. When the content of the database is changed, modifications are reflected in the cooperative cache.

CoopSC handles the execution of n-dimensional select-project queries. Similarly with the approach presented in [31], the local cache is organized into disjoint *semantic regions*. A semantic region is defined as a set of tuples and a constrained formula which determines the common property of the tuples. Clients interrogating a specific database server form the P2P overlay network, which is used for indexing these semantic regions.

Semantic regions are stored, in-memory, by the Cache Manager [91]. The Cache Manager also implements a replacement policy which is orthogonal to the CoopSC approach. Each client manages its local cache entries independently. As a result, popular entries will be automatically replicated to multiple clients and thus the scalability of this approach is increased.

This chapter is organized as follows: the basic concepts related to the CoopSC approach, which are used throughout this chapter, are introduced in Section 3.1 . The query rewriting process is described in Section 3.2. This is followed by a detailed description of the design of the distributed index in Section 3.3. The update mechanism is presented in Section 3.4 while Section 3.5 presents this chapter's summary and some concluding remarks related to the design of the CoopSC approach

3.1 Basic Concepts

Based on the general semantic caching approach, introduced in [31], this section presents the main structures used in the CoopSC approach and also formally defines the possible relations between semantic regions and queries. These definitions are further used when describing the query rewriting algorithms.

Figure 3.2 describes the main structures which are used for representing semantic regions and queries. The *SemanticRegion* structure contains a unique identifier, the name of the table, the set of fields, the predicate which describes it, and the set of tuples which determines the content of region. A *query* is defined by the name of table, the set of fields and the predicate.

Based of structures presented in Figure 3.2, is what follows the concepts of intersection and subsumption between queries and regions will be clearly defined.

Definition 1 *Let r be a semantic region and q a query. r and q are said to* vertically intersect *if* $r.table = q.table$ *and* $r.fields \cap q.fields \neq \emptyset$.

Example 1 *Let:*

$r_1 = (10, wisconsin, \{unique1, unique2, two\}, unique1 < 10, \{...\})$

$r_2 = (11, wisconsin, \{unique2, four\}, unique1 < 10, \{...\})$

two semantic regions and

$q = (wisconsin, \{unique1, two\}, unique1 < 40)$

a query.

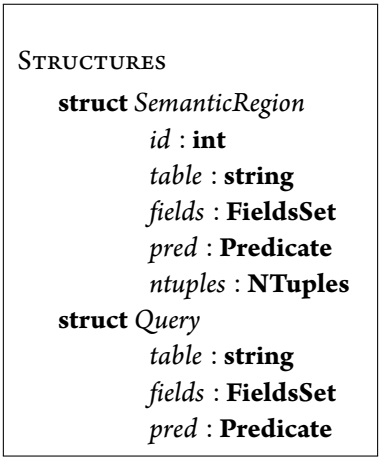

Figure 3.2: CoopSC Structures

r_1 *and* q vertically intersect, *while* r_2 *and* q *do not because their fields do not intersect.*

Definition 2 *Let r be a semantic region and q a query. r and q are said to* horizontally intersect *if* $r.table = q.table$ *and the predicate* $q.pred \wedge r.pred$ *is satisfiable.*

Example 2 *Let:*

$r_1 = (10, wisconsin, \{unique1, unique2\}, unique1 < 10, \{...\})$

$r_2 = (11, wisconsin, \{unique2, unique2\}, unique1 > 100, \{...\})$

two semantic regions and

$q = (wisconsin, \{unique1, two\}, unique1 < 40)$

a query.

r_1 *and* q horizontally intersect, *while* r_2 *and* q *do not because the predicate* $(unique1 > 100) \wedge (unique1 < 40)$ *is not satisfiable.*

Definition 3 *Let r be a semantic region and q a query. r and q are said to* intersect *if they* vertically intersect *and* horizontally intersect.

If a semantic region R intersects a given query Q, it can be used for partially answering Q. Query will be split in a sub-query that can be

answered using R and sub-queries for which R does not provide relevant tuples.

Definition 4 *Let r_1 and r_2 be two semantic regions. r_1 and r_2 are said to be* disjoint *if $r_1.table \neq r_2.table$ or $r_1.fields \cap r_2.fields = \emptyset$ or the predicate $r_1.pred \wedge r_2.pred$ is not satisfiable.*

Example 3 *Let:*

$r_1 = (10, wisconsin, \{unique1, unique2, two, four\}, unique1 < 10, \{...\})$

$r_2 = (11, wisconsin, \{unique1, unique2, two\}, unique1 > 100, \{...\}$

$r_3 = (12, wisconsin, \{unique2, four\}, unique1 > 0, \{...\}$

three semantic regions.

r_1 and r_2 are disjoint, while r_1 and r_3 are not because $(unique1 < 10) \wedge (unique1 > 0)$ is satisfiable.

Definition 5 *Let r be a semantic region and q a query. It is said that r* vertically subsumes *q if $r.table = q.table$ and $r.fields \supseteq q.fields$.*

Example 4 *Let:*

$r_1 = (10, wisconsin, \{unique1, unique2, two, four\}, unique1 < 10, \{...\})$

$r_2 = (11, wisconsin, \{unique1, unique2, two\}, unique1 < 10, \{...\}$

two semantic regions and

$q = (wisconsin, \{unique1, four\}, unique1 < 40)$

a query.

r_1 vertically subsumes *q, while r_2 does not because $r_2.fields \not\supseteq q.fields$.*

Definition 6 *Let r be a semantic region and q a query. It is said that r* horizontally subsumes *q if $r.table = q.table$ and $q.pred \Rightarrow r.pred$.*

Example 5 *Let:*

$r_1 = (10, wisconsin, \{unique1, unique2\}, unique1 < 100, \{...\})$

$r_2 = (11, wisconsin, \{unique2, unique2\}, unique1 < 50, \{...\})$

two semantic regions and

$q = (wisconsin, \{unique1, two\}, unique1 < 70)$

a query.

r_1 horizontally subsumes *q, while r_2 does not because $(unique1 < 70) \not\Rightarrow (unique1 < 50)$.*

Definition 7 *Let r be a semantic region and q a query. It is said that r* subsumes *q if r* vertically subsumes *q and r* horizontally subsumes *q.*

If region *R* subsumes query *Q*, *Q* can be answered completely using the content of *R*. Thus, subsumption is a much stronger condition than intersection.

3.2 Query Rewriting

The query rewriting process (illustrated in Figure 3.3) determines parts of a given query that can be answered using the local cache (*probe*), caches of other clients (*remote probe*) or the database server (*remainder*) and the way in which they are combined in order to return the final query result. This process is executed by a component, running on client side, called *Query Rewriter*. The result of a query rewriting process is a *query plan tree*, which describes how query is to be executed. Initially, the query rewriting checks entries stored in the local cache (*Local Rewriting*). Afterwards, the distributed index is interrogated in order to determine remote cache entries which can be used for answering given query (*Distributed Rewriting*).

This section will, first, describe the structure of *query plan trees*. Afterwards, the local and distributed rewriting process will be presented.

3.2.1 Query Plan Tree

As mentioned, the result of a query rewriting process is a *query plan tree* (exemplified in Figures 3.5 and 3.6). Its leaves refer to semantic regions (stored locally or remotely) or sub-queries which are to be executed by the database server.

A *query plan tree* contains types of nodes for executing union and join operations, selecting tuples from the local cache entries (*SelectProject*), returning the content of specified region (*Region*), executing given query on server (*Remainder*) and returning result of a query plan tree executed on a different CoopSC (*RemoteProbe*).

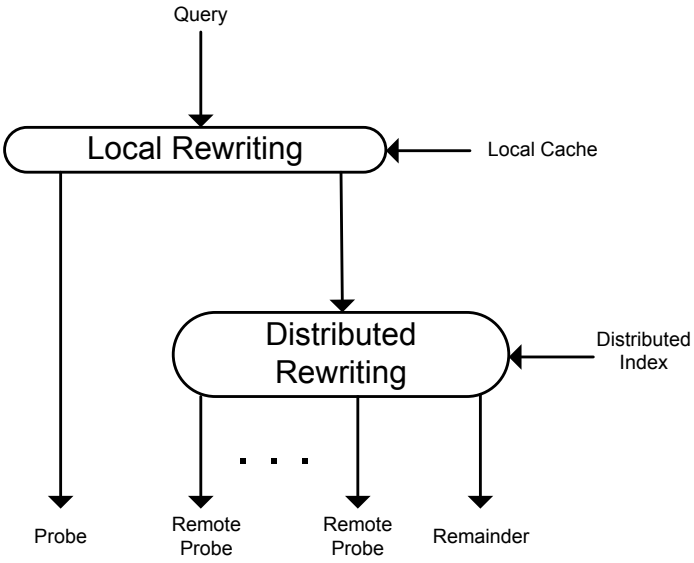

Figure 3.3: Query Rewriting

3.2.2 Local Rewriting

The *Local Rewriting* process scans the local cache and determines which semantic regions can be used for answering a given query. The result of the local rewriting is an initial *query plan tree* which only contains references to the local cache or the database server (exemplified in Figure 3.5).

Each region is compared with a given query. Figure 3.4 illustrates the possible relations between a given query and a current semantic region. The vertical and horizontal axes are used to illustrate the concepts of vertical and horizontal intersections, introduced in Section 3.1. In this figure, the red-bordered rectangles represent parts of given which are not contained in current region and need to be answered using other regions. When comparing given query with a region, the following situations emerge:

a) region and query do not *intersect* (c.f., Definition 3); current region is not used for answering query.

b) region *subsumes* (c.f., Definition 7) query; query can be completely answered using current semantic region.

c) region *vertically subsumes* (c.f., Definition 5) query and *horizontally intersects* (c.f., Definition 2) it; query is split into two sub-queries; one can be answered using current region, while for the other the local rewriting will continue using the following regions.

d) region *horizontally subsumes* (c.f., Definition 6) query and *vertically intersects* (c.f. Definition 1) it; query is split into two sub-queries; one can be answered using current region, while for the other the local rewriting will continue using the following regions.

e) region both *horizontally* (c.f., Definition 2) and *vertically intersects* (c.f. Definition 1) query; query is split into three sub-queries; one can be answered using current region, while for the other two the local rewriting will continue using the following regions.

Thus, an initial query plan tree is constructed by comparing comparing queries with each region, determining the intersection, and continuing the process with that parts of given queries which are not part of this intersection.

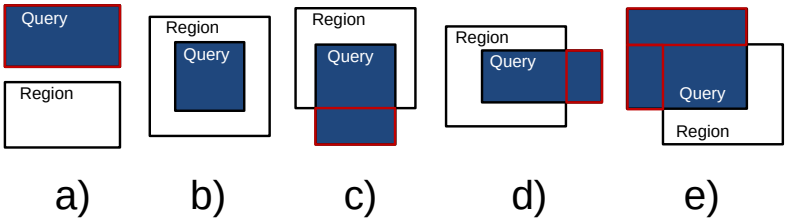

Figure 3.4: Region/Query Overlapping

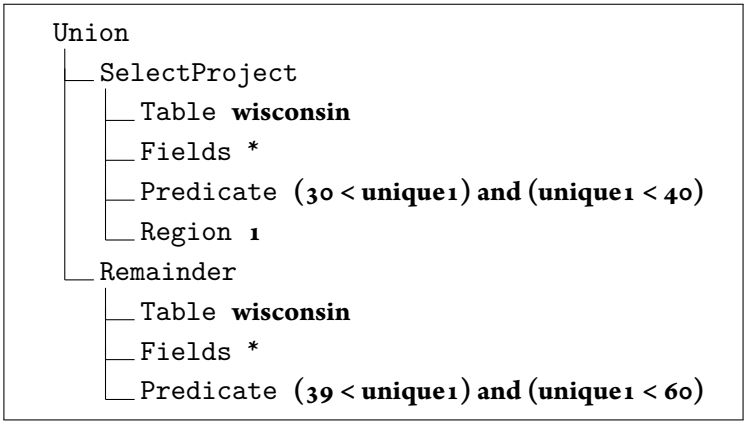

Figure 3.5: Local Query Plan Tree

3.2.3 Distributed Rewriting

As illustrated in Figure 3.3, *distributed rewriting* uses the distributed index in order to determine which remote semantic regions can be used for answering given query. The *query plan tree*, generated during *local rewriting* is modifying by replacing *Remainder* nodes with results of interrogations sent to *distributed index*. These results can refer to semantic regions stored by other clients. Figure 3.6 illustrates a possible result of distributed rewriting.

3.3 Distributed Index

This section describes the distributed structure that is used for indexing semantic regions. For visualization purposes, only double attribute selections are considered, but, afterwards, the way in which this approach can be generalized for multi-attribute selections is presented.

As mentioned in the beginning of the section, semantic regions are defined by a set of tuples and a predicate. Under the given two-dimensional assumption, the predicate is a double attribute selection (Example: $10 < lat$ and $lat < 20$ and $20 < long$ and $long < 30$).

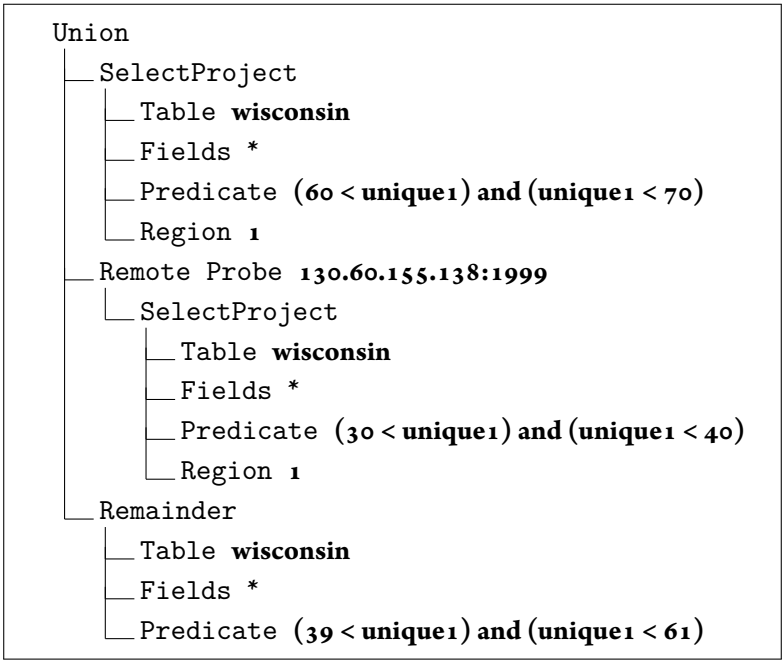

Figure 3.6: Distributed Query Plan Tree

Queries are also double attribute selections (Example: select * from earthquakes where 10 $<$ *lat* and *lat* $<$ 20 and 20 $<$ *long* and *long* $<$ 30). Double attribute selection predicates can be represented as sets of non-overlapping axis-aligned rectangles (Example: {(10, 10, 20, 30), (40, 50, 80, 90)}). Rectangles are represented with the coordinates of their top-left and bottom-right corners. This representation will be used for both semantic regions and queries.

The distributed index must be able to index semantic regions. Removing regions from index shall also be supported. Furthermore, given a query Q, the distributed index must return a *query plan tree* that contains references to semantic regions stored in different CoopSC clients and minimizes the part of query which is answered by the database server.

The distributed index is based on the P2P index described in [86], which adapts the classic MX-CIF quad trees [81] in order to be stored on top of a P2P overlay. CoopSC implements this approach and also integrates the distributed rewriting algorithm (c.f., Figure 3.10) into the general distributed index structure introduced in [86].

The two-dimensional square-based area is, recursively, divided into four equal-sized square blocks until a given *fundamental maximum level*, f_{max} is reached. Each square is associated with a node from the P2P overlay. The association between squares and peers is done by applying a hash function on coordinates of squares' center points and selecting, for each square, the peer that has the closest ID to the hash value. Each semantic region is indexed in the square of *minimum size* (thus, maximum level) that contains given region.

Figure 3.7 exemplifies the quad tree space division and the way in which four give semantic regions are indexed. Regions R_1 and R_4 are indexed in the root node, since they are not contained in any child square. R_2 is totally contained in the top-right level 2 sub-square, thus is indexed in the corresponding level 2 node. R_3 is indexed in a level 3 node.

As described in [86], for performance and reliability reasons, the root node is not stored in the P2P overlay. Only nodes at a level higher or equal to a given *fundamental minimum level*, f_{min} are considered. Thus, when implemented distributively on top of a P2P overlay MX-CIF quad trees are transformed into *forests*. It is assumed that f_{min} is chosen in such a way that every semantic regions is contained in a square associated with a node of level f_{min}.

Figure 3.8 and 3.9 present the pseudo-codes for adding and removing semantic region to/from distributed index. As mentioned, each region is added to the quad node of maximum level that contains it. Methods *sendAddRegion* and *sendRemoveRegion* route request through the P2P overlay until the node associated with the specified *Quad* is reached.

Figure 3.10 contains the pseudo-code for the algorithm that handles rewriting requests within distributed index. An initial query rewriting, similar with local rewriting, is performed using indexed semantic re-

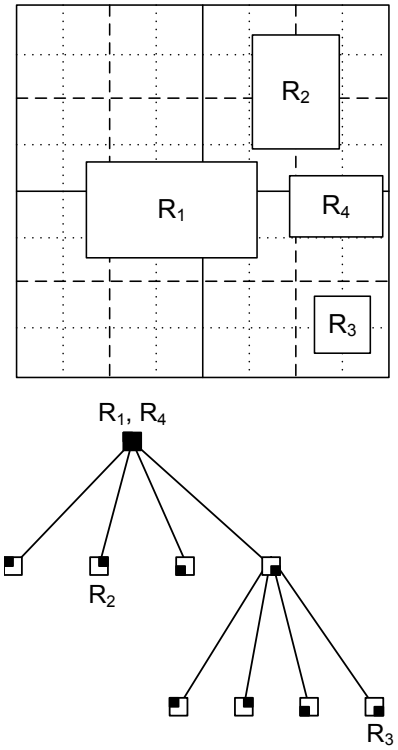

Figure 3.7: MX-CIF Quad Tree Example

gions. Afterwards, the rewriting process continues with children quad nodes.

```
ADD-REGION(QN : QuadNode, R : Region)
    for child : QN.children()
        do if child.contains(R)
              then sendAddRegion(child, R)
                   return
    QN.regions.add(R)
```

Figure 3.8: Distributed Index: Add Region

```
REMOVE-REGION(QN : QuadNode, R : Region)
    for child : QN.children()
        do if !child.empty() ∧ child.contains(R)
            then sendRemoveRegion(child, R)
                 return
    QN.regions.remove(R)
```

Figure 3.9: Distributed Index: Remove Region

```
REWRITE(QN : QuadNode, Q : Query)
    n ← rewrite(Q, QN.regions)
    for r : n.remainders()
        do result ← ∅
            for c : QN.children()
                do if c.intersects(r.query)
                    then
                        m ← sendRewrite(c, r.query)
                        results.add(m)
            r ← Union(results)
    return n
```

Figure 3.10: Distributed Index: Rewrite

The distributed index can be adapted to n-dimensional selections by dividing the n-dimensional space into 2^n equal size quads. For single attribute selections, quads are reduced to intervals.

3.4 Updates

When the content of the database is changed, modifications must be reflected in the cooperative cache. Handling updating efficiently presents the following challenging issues: a) not all modifications are generated directly by clients; the database server can have active compo-

nents which perform changes as a result of different events; b) the update mechanism must avoid combining region that pertain to different database snapshots.

```
EXECUTE(Q : Query)
    quads ← query.getIntersectedSquares(f_update)
    before ← database.getTimestamps(quads)
    plan ← rewrite(Q, before)
    result ← plan.execute()
    after ← database.getTimestamps(quads)
    if before ≠ after
       then return database.execute(Q)
       else  return result;
```

Figure 3.11: Update Handling

The following example illustrates a scenario when combining regions that originate from different snapshots causes inconsistencies: client A executes Q_1: "select * from persons where $20 < age$ and $age < 40$" and caches its result in region R_1. Afterwards, client B updates the age of a person from 25 to 50. Finally, A execute Q_2: "select * from persons where $20 < age$ and $age < 60$" which is split in a probe which returns region R_1 and a remainder which executes "select * from persons where $39 < age$ and $age < 60$" on server-side and returns result. The updated person will be present both in region R_1 and also in the remainder, because the modification was performed after the execution of Q_1 and thus, the final result will be inconsistent.

CoopSC handles updates with a cooperation from the database server. An active database server component was developed in order to handle the execution of update, insert, and delete SQL statements using triggers. This component uses the same quad space division as the distributed index which was presented in the previous section. For each quad from a given *fundamental update level*, f_{update} ($f_{min} \leq f_{update} \leq$

f_{max}), database server stores a virtual timestamp which is initialized with 0. These timestamps are incremented when modification are performed to tuples pertaining to particular quads. Semantic regions are augmented with virtual timestamps of quads they intersect at the moment of retrieval from database. Referring to the example from figure 3.7 and assuming that $t_{update} = 2$, R_1 will store four timestamps values, R_2 one timestamp, R_3 also one, and R_4 two timestamps.

Before rewriting a new query, client asks the database server for the virtual timestamps of the quads that intersect given query. The rewriting process will not use entries for which some virtual timestamps are older than the ones returned by server. If such entries are found, they are also discarded in order to save storage space. These timestamps are also used during distributed rewriting in order to only consider up-to-date remote semantic region challengings and to discard old ones.

On one hand, an advantage of this approach is that queries results are always up-to-date. On the other hand, this solution can discard entries that are still valid. A modification performed on a single tuple, which pertain to a particular quad causes invalidation of all entries which intersect that quad. Increasing the fundamental update level can reduce the number of valid entries which are discarded, but, in the same it also enlarges the number of virtual timestamps stored in distributed index and regions.

Figure 3.11 contains the pseudo-code of the update mechanism within the general query execution algorithm. Initially, quads of level f_{update} which intersect given query are determined. The timestamps of these squares are then returned from the database server. The query rewriting process is then executed and a query plan tree returned. The query plan tree is executed and a result returned. Afterwards, the timestamps values of query's squares are determined again and compare with the original values. If differences are noticed, the result that uses the cache is discarded, because at least a modification occurred during query rewriting or query plan execution and system can not guarantee that result contains only tuples from a single database snapshot.

Since clients are not directly involved when executing update statements, modifications performed outside the CoopSC context are also handled correctly. Moreover, the use of the virtual timestamps makes sure that all semantic regions which are integrated belong to the same database snapshot which guarantees that the final query result is consistent. Thus, both challenges related to handling updates, described in the beginning of this section, are solved by the CoopSC approach.

3.5 Chapter Summary

This chapter describes the CoopSC cooperative semantic caching approach which aims to improve the performance of SQL n-dimensional range interrogations in distributed environments. The main concepts and processes related to CoopSC were introduced and presented in details. The design of the distributed query rewriting algorithm and of the distributed index are described and exemplified. The structure of *query plan trees*, which are returned by the query rewriting process, is outlined. Moreover, efficient mechanism were designed for handling update statements.

Thus, by introducing and describing the CoopSC approach, this section shows that a cooperative semantic caching approach is technical feasible and that, within the general approach, update statements can be handled in a way that guarantees consistency. Based on this approach, a CoopSC system was developed (c.f. Chapter 4) and evaluated (c.f. Chapter 6).

4

The CoopSC Architecture

This chapter describes the architecture of the CoopSC system which handles the execution of n-dimensional range select-project queries and also supports the SQL statements which modify the content of database (i.e, *insert*, *delete*, *update*). The system was fully implemented (c.f., Chapter 5) and evaluated (c.f., Chapter 6). The newly developed architecture is based on the approach presented in Chapter 3, and aims to integrate all the CoopSC-related features in a unified system. The CoopSC architecture works with relational database systems. The local cache is organized into disjoint semantic regions, as defined in Chapter 3. Clients interrogating a specific database server form the P2P overlay network, which is used for indexing semantic regions.

4.1 Overview

The CoopSC architecture must allow the homogeneous embedding of all CoopSC functionalities into a single integrated system. Moreover, this architecture must also make sure that all of these features can be implemented in an efficient way.

Most of the CoopSC features are running on the client side. Thus, cache entries must be stored and retrieved efficiently using a storage component. A P2P overlay architecture must be used in order to index semantic regions and to efficiently determine relevant cache entries. The query rewriting process, which accesses the distributed index and the local cache, should also be embedded in the new CoopSC system. Components must be designed for allowing the execution of sub-queries and the integration of their partial results into a single query result. Since clients should be able to connect to each other and execute remote subqueries, a classic client-server interaction must be supported and designed in an efficient way. Thus, each CoopSC-enabled node must be able to act as a server for all of its peers.

Because the update handling mechanism is done with a cooperation from the database server, the CoopSC architecture also contains some update-related components which run on server side. These components interact with the database trigger mechanism and are called when modifications are performed on tables for which the CoopSC approach was enabled. These server components also store the general CoopSC metadata which is read by all clients during the system initialization phase.

4.2 Components

The components of the CoopSC architecture, utilizing the P2P overlay functionality, are illustrated in Figure 4.1. In what follows, these CoopSC architectural components are described outlining their main features and also the way they interact with each other.

4.2.1 Client Components

Because CoopSC aims to allow clients to exchange cache entries in a cooperative way, most of the CoopSC-related functionality runs on client side. An API (Application Programming Interface) is provided for allowing applications to use the CoopSC system. A *query rewriting* com-

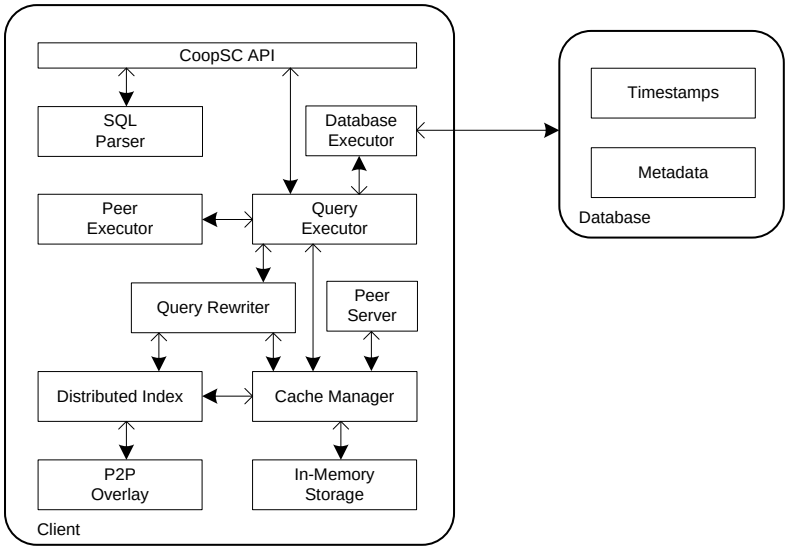

Figure 4.1: CoopSC Architecture

ponent determines which components of a given query are to be executed on local cache, database server or remote peers. Components are also provided for executing sub-queries on database servers and remote clients. In what follows, the CoopSC client-side components are described in details.

CoopSC API

The *CoopSC API* (c.f., Figure 4.1) is a programming interface that allows writing applications that use the CoopSC architecture. This component allows CoopSC-based applications to create CoopSC connections, execute SQL statements, and iterate through results of query executions.

In order to allow existing Java applications to use CoopSC, a CoopSC JDBC (Java Database Connectivity) [77] driver was also developed. The JDBC API-called are mapped to the standard CoopSC-API using the JNI (Java Native Interface) [45] technology.

SQL Parser

The *SQL parser* (c.f., Figure 4.1) component parses the sub-set of the SQL language which corresponds to the statements that are handled by the CoopSC system. Thus, this parser handles selection queries for which the predicate is n-dimensional range selection. Other query types are ignored by the parser and send for execution directly to database server.

Query Executor

The *Query Executor* (c.f., Figure 4.1) component handles the execution of queries. Using the *Query Rewriter*, it first splits a given query into probe, remote probes and remainder sub-queries. These sub-queries are integrated into a single *query plan tree*.

Furthermore, it executes each sub-query, in parallel, by accessing the local cache or by sending it for execution to either a different client or to the database server. After the execution of these sub-queries, the *Query Executor* integrates their result sets and returns the final result of the query.

Since the relational database model is based on a the set theory, the order of tuples returned by SQL select statements (without an *order by* clause) is not relevant. Thus, integrating different result sets can be performed efficently without needing to establish a fixed order of integration.

Before and after each query execution the *Query Executor* determines, by accessing the database server, the virtual timestamps of the quads which contain the given query. As described in the previous chapter, these timestamps are used for making sure that old cache entries are not used when generating new results set.

Query Rewriter

The *Query Rewriter* (c.f., Figure 4.1) determines which parts of a given query are to be returned from local cache, remote cache or the database

server. The result of the query rewriting process is a *query plan tree* which contains parts that are to be executed locally, on the database server or on remote clients.

In the first step, the *Query Rewriter* determines the *probe*, which is the part of the query that is available in the local cache. This is accomplished by scanning local semantic regions and checking if they overlap with the given query. This first step also returns the *local remainder*, which represents the portion of the query which is not available in the local cache.

In order to determine the *remote probes* and the *remainder*, the distributed index is interrogated with the local remainder.

Database Executor

The *Database Executor* (c.f., Figure 4.1) sends *remainder* sub-queries to be executed on database servers and returns the corresponding result sets. These results sets are then used by the *Query Executor* and combined with other sub-queries results in order to determine the final query result. The communication between this component and the database server is based on the well-established SQL protocol.

Peer Executor

The *Peer Executor* (c.f., Figure 4.1) component handles the execution of *remote probe* sub-queries which return data from remote caches of other clients. This component receives a *query plan tree* from the *Query Executor* component. This query plan tree is sent to a remote *Peer Server* for execution. The result of the execution is then sent back and stored locally.

In order to improve performance, single connection to other clients are reused by multiple query executions and stored in a *connection pool*.

Cache Manager

The *Cache Manager* (c.f., Figure 4.1) component manages the storage of semantic regions and implements the LRU (Least Recently Used) re-

placement policy. LRU is motivated by the concept of temporal locality: a recently used entry has a good chance to be used in the near future.

The distributed index must be synchronized with the content of clients' local caches and, thus, when a semantic region is added or removed from the cache, the distributed index component is notified and the structure of index is updated.

Distributed Index

All semantic regions are indexed in a *Distributed Index* (c.f., Figure 4.1). The purpose of the distributed index is to make the query rewriting process efficient. For each query given as input, the distributed index component returns a list of semantic regions that semantically overlap the query and minimize the portion of query that must be executed by the database server. The design of the distributed index is described in Section 3.3. Thus, this component's interface must provide methods for indexing semantic regions, removing entries from the distributed index and for executing the distributed query rewriting process.

Peer Server

The *Peer Server* (c.f., Figure 4.1) handles requests of locally cached data received from other CoopSC clients. The requests contain *query plan trees* which refer local semantic regions. These query plans are executed and results are returned to caller remote clients.

The design of this component must allow multiple query execution to be executed in parallel because multiple remote peers can request data from the local cache in the same time.

In-Memory Storage

The *In-Memory Storage* component (c.f., Figure 4.1) handles the storage of semantic region on client-side. This storage component is directly managed by the *Cache Manager* component which determines which semantic regions are stored or removed from the cache. The *In-Memory*

Storage provides a simple interface which has methods for adding and removing cache entries.

P2P Overlay

The *P2P Overlay* (c.f., Figure 4.1) layer implements a DHT (Distributed Hash Table) [13] functionality which is used by the *Distributed Index* component in order to index semantic regions. In such an approach, a hash function is applied to each entry which is about to be stored and the node with the closest ID is chosen for storage. The P2P overlay utilizes the SHA1 [38] hash function.

This hash function is applied when indexing a cache entry in order to determine which node from the overlay is chosen for storing the index of that cache entry. The description of the cache entry (i.e., selection predicate and attributes) is serialized and the hash function is applied to this serialized data.

The P2P overlay implements a routing algorithms (e.g., Kademlia [13]) which routes a particular (*key*, *value*) entry to the right node. When indexing semantic regions, the *key* is assumed to be formed from the coordinates of the quad that totally contain given region, while the *value* contains the description of the query which generated the semantic region, the value of the relevant virtual timestamps, the address of the nodes that stores it, and the ID of the region.

Such an approach increases the scalability of the system since the load of storing and interrogating the distributed index is divided to all CoopSC-enabled clients. Unfortunately, this solution has also an important drawback: if some nodes fail, the entries stored of that nodes are lost and, thus, the corresponding semantic regions can not be used by other clients.

On one hand, in the context of CoopSC, even if some entries from the distributed index are completely lost, the system would not return wrong or incomplete results since the logically-centralized database server has the complete set of data. On the other hand, if many such en-

tries are lost the general performance degrades since the database server can become overloaded.

In order to improve reliability multiple nodes can be used for the storage of a particular entry, based on a replication factor. Thus, given an entry, the system finds the R (replication factor) nodes with the closest IDs to the hash value and stores entry on all of them. If some such nodes fail, other can take over.

4.2.2 Database Server Components

The CoopSC server components contain the update mechanism, and also provide a unified access to the CoopSC metadata. The update mechanism is designed as a PostgresSQL trigger component which is called when modifications are performed in CoopSC-enabled database relations. In what follows, these CoopSC server components are presented and described.

Update Trigger

The *update trigger* component is automatically called when modifications (i.e. *update*, *delete* and *insert* statements) are performed on tables for which CoopSC was configured in order to increment the virtual timestamps which are affected by these modifications. Such a trigger is automatically created for each CoopSC-enable database table.

Timestamps

The *Timestamps* component (c.f., Figure 4.1) stores virtual timestamps related to tables and fields for which the CoopSC approach was configured.

A virtual timestamps is initialized with 0. For every modification performed, its value is incremented by 1 by the *Update Trigger* component. These timestamps are stored in database tables which are created automatically by the CoopSC system.

Metadata

The *Metadata* component (c.f., Figure 4.1) stores information about tables and attributes for which the CoopSC is used. All numerical parameters related to the *Distributed Index* and to the update handling mechanism are also kept (i.e, minimum level, maximum level, update level). The metadata is read by all CoopSC clients during the initialization phase.

Storing the CoopSC metadata on the server side is motivated by the fact that it is important that all clients have the same view of the configuration of the CoopSC system. For example, all clients must know for which table CoopSC was enabled and which are the values of the configuration parameters. Since the metadata is stored in relational table, the communication between the client components and the Metadata component is done using SQL interrogations.

4.3 Interactions

Figure 4.1 also illustrates the interactions between the CoopSC architectural components. Internally, all these components communicate in a typical request-reply matter. This two types of interaction are visualized in the figure by using arrows of different size (larger arrows are used of requests, smaller for replies).

This section describes the interactions between the CoopSC architectural components in the context of the three main execution scenarios: a) a new locally initiated query execution; b) execution of a remote probe which was received from a different peer; c) handling modifications performed on database server. These three scenarios were chosen because they express the core functionality of the CoopSC architecture and embed all component-to-component individual interactions.

4.3.1 Query Execution

The message sequence diagram from Figure 4.2 illustrates the interactions between the CoopSC architectural components when executing a new query.

Initially, the query is parsed using the *SQL Parser* component. Afterwards, the *Query Executor* is called in order to execute the given query. The *Query Rewriter* component then executes the two-phase query rewriting process. The local rewriting process reads the local *Cache Manager* while the distributed rewriting process accesses the *Distributed Index* component. The returned sub-queries are then executed by accessing the *Cache Manager* or by sending them for execution to either the *Peer Executor* or to the *Server Executor*. Lastly, the final query result is computed by integrating the partial results of these sub-queries executions.

4.3.2 Remote Execution

Figure 4.3 illustrates the interactions between the CoopSC architectural components when a new remote probe sub-query is received from a remote peer (*Caller*) and is to be executed by the current node (*Callee*).

The sub-query is first sent by the *Peer Executor* component from the *caller* node to the *Peer Server* component of the current node. The communication is based on the TCP/IP (Transmission Control Protocol/Internet Protocol) transport protocol. The use of TCP/IP is motivated by the need of reliable transmission when returning remote query results. The sub-query is then executed by accessing the local semantic cache, and the result of this execution is sent back to the *caller* CoopSC node.

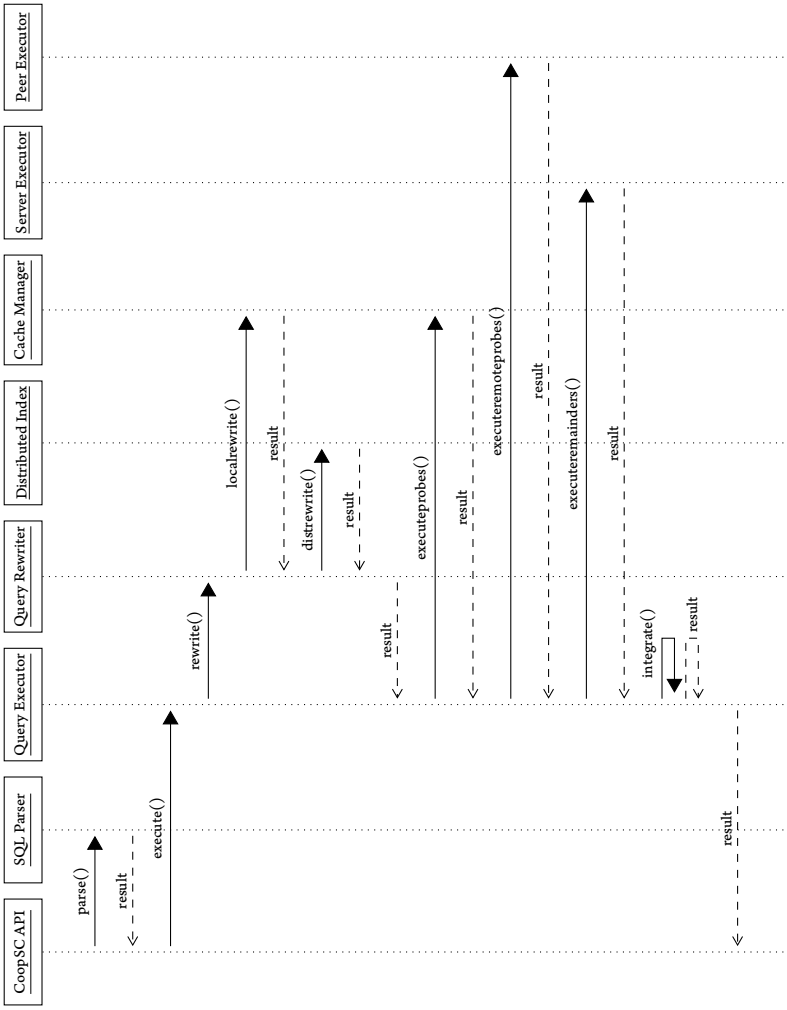

Figure 4.2: Query Execution Message Sequence Diagram

4.3.3 Update Handling

Figure 4.4 illustrates the interactions between the CoopSC server-side components when a modification is performed in a table for which the CoopSC approach was configured.

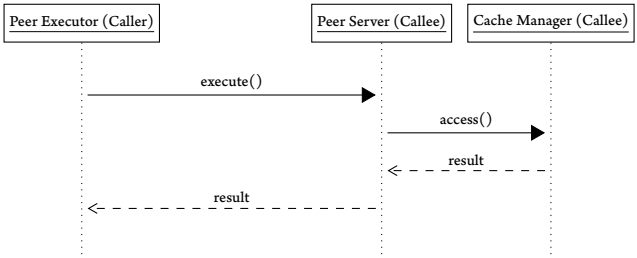

Figure 4.3: Remote Peer Execution Message Sequence Diagram

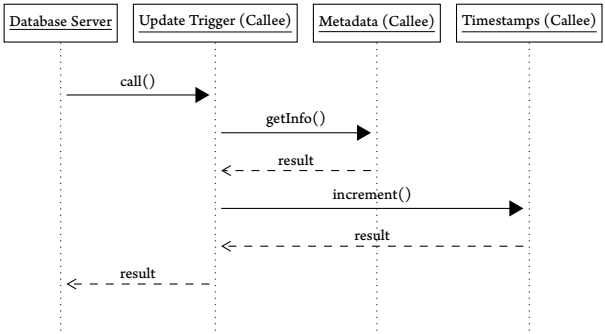

Figure 4.4: Update Handling Message Sequence Diagram

The CoopSC *Update Triggers* is initially called as a result of a modification which was performed in a CoopSC-enabled table. The trigger then accesses the *Metadata* components in order to determine the parameters with which the *Distributed Index* was configured. Finally, based on this information, an entry from the *Timestamps* table is incremented.

4.4 Protocol Design

This section describes the design of the protocols which are used during the communication between the CoopSC-enabled clients and the database server. Since CoopSC is a peer-to-peer system, two types of interactions are analyzed: client-database, and client-client.

4.4.1 Client-Database Interactions

Since the database server is assumed to be a relational database server, the client-database communication protocol is using the SQL language. Based on the component interactions described in the previous section, there are three types of messages exchanges between clients and the database server.

Metadata Retrieval This interaction only happens in the beginning, during the initialization phase. It consists in the execution of single select statements which returns the full content of the metadata table, which it is assumed to be of reduced size (e.g.,*select * from coopsc.metadata*).

Timestamps Retrieval Before execution a CoopSC-supported query, the timestamps of quads which intersect given query must be determined. Thus, clients send SQL interrogations which select from the timestamps table only the relevant values.

Query Execution Queries are sent for execution to the database server and their results are returned back to clients in a typical SQL fashion.

4.4.2 Client-Client Interactions

The client-client interactions consist on both direct communications between two clients and also on the messages exchanged using the P2P overlay. When P2P messages are exchanged, destination node is not specified by sender but it is selected by the overlay system based on the value of a hash function. In what follows, the protocols used for communication during the interaction scenarios described in the Section 4.3 are presented. Initially, the protocol used during the execution of a remote query is described (*Remote Execution*). Afterwards, the protocol used by the distributed is presented (*Distributed Index*).

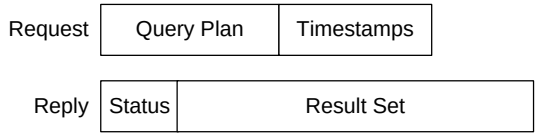

Figure 4.5: Remote Execution Message Format

Remote Execution

In the *remote execution* scenario, a client sends a sub-query for execution to a remote peer and the result of this query execution is returned. The message sequence diagram from Figure 4.3 shows the interaction between the *caller* and the *callee* clients.

Figure 4.3 illustrates the structure of messages used when executing a remote query. When sending a request *caller* has to specify a *query tree plan* which is to be executed, and a list of *virtual timestamps* which are used for making sure the only up-to-data semantic regions are utilized. A reply contains a status field which indicates if the execution was successful and a result set (i.e., a list of tuples). The implementation of the architecture must decide how to serialize the structures sent and received during this interaction.

Distributed Index

Clients also communicate with each other in order to form and maintain the distributed index, which was described in Section 3.3. Special messages are exchanged for adding entries, removing entries and returning query rewritings. A distributed index entry contains information needed to access a specific region (i.e. region and client IDs), description of that region (i.e. query which was used for its generation) and also values of virtual timestamps of quads that intersect given region. Each message is routed to a specific node of the overlay based on the returned value of a hash function, in a typical DHT matter.

Figure 4.6 illustrates the structure of messages sent/received for creating and maintaining the distributed index. The first field represents the

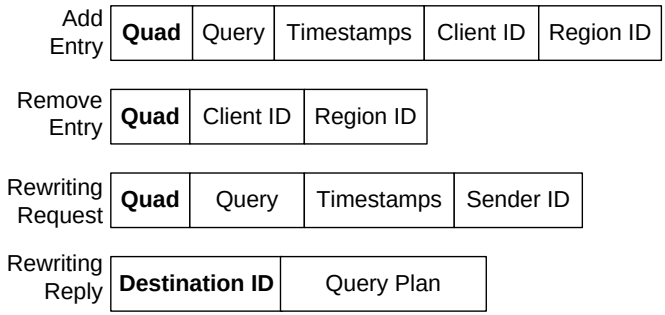

Figure 4.6: Distributed Index Message Format

value which is used for routing the message in order to reach the destination destination.

When a region is indexed in the distributed index, the following arguments are specified in the message (c.f., Figure 4.6): quad which totally contains region (*quad*), query which was used for generating region (*query*), virtual timestamps(*timestamps*), ID of client that stores region (*Client ID*), and region's ID (*Region ID*). A hash function is applied to the *quad* argument and the peer with the closest ID to hash value is chosen for indexing given region. A good hash function must be chosen in order to determine a good distribution of the cache entries to all peers. Removing an entry involves sending a message with quad that contains given entry (*quad*), IDs of peer that stores it (*Client ID*) and of given region (*Region ID*). Similarly, this message is routed to the right node based on value of the *quad* argument.

A query rewriting request (c.f., Figure 4.6) contains destination quad (*Quad*) query which is about to be rewritten (*Query*), current virtual timestamps (*Timestamps*), and identity of sender (*Sender ID*). Timestamps are needed in order to make sure that old entries are not returned during the rewriting process. The identity of the sender is also necessary in order to know where to send the result. A rewriting reply message contains destination node (*Destination ID*) and a query plan which refers remote semantic regions.

4.5 Use Case 1: CoopSC within Cloud Environments

This section investigates the use of the CoopSC architecture with-in real-life cloud computing environments, emphasizing the economic dimension of the potential scenarios. These scenarios are meant to serve as motivating examples for the general CoopSC approach.

Nowadays, cloud computing environments (e.g., Amazon EC2 [10], Rackspace [76]) have become an important technology which delivers computing resources (CPU, bandwidth, storage) as a service. End-users do not need to have information about the physical location of these resources, which can be scaled up or down, in an elastic matter, depending on the real-time demand [7]. End-users are charged based on the usage of these resources. Thus, optimizing the use of cloud-based computing resource has now also an important economic importance.

4.5.1 Background

One of the key purposes of caching mechanisms is to reduce the volume of transferred data. Less transferred data can be translated in less costs to maintain an application. Therefore, in the scope of this thesis, it is important to analyze which providers or technologies enable an optimum gain to deploy CoopSC. Cloud Computing can be considered as a technology that enables solutions as CoopSC with an optimized spending since the "pay-per-use" concept [7] is embedded in the cloud providers business model. Just paying for the data that is actually transferred between nodes (i.e., server and clients) has clear advantages over the traditional fashion, when customers used to pay a monthly fixed amount independently of how much traffic was spent. However, cloud providers may present different charging schemes, mainly related to Infrastructure-as-a-Service (IaaS) [7] products. It is important to analyze each of the charging schemes before deploying any solution into the Cloud.

In the past, Amazon EC2 [10] did not charge for data transfers of two virtual machine instances in the same availability zone. The availability

zone is a region that customer defines to host its instances in the moment of the Amazon's instance creation. Nowadays,e.g., Amazon charges in and out data transfers independently of the instance's zone [11], meaning that any inbound or outbound traffic will be charged on each virtual node. Rackspace [76] offers a product called Cloud Servers which is very similar to Amazon EC2. Among some technical and feature differences between them, the charging scheme differs related to data transfer between two virtual instances: if the customer does not transfer data using the network interface which has a public IP assigned, such transfer is not charged at all. GoGrid [43] can be considered as a midpoint taking into consideration on how Amazon and Rackspace charges for data transfer. GoGrid does not charge per any inbound traffic into deployed instances, however it charges per any outbound traffic. The examples illustrated above may have advantages and/or disadvantages depending on how the traffic is generated considering the application employed.

4.5.2 Scenarios

The CoopSC architecture reduces the amount of data transferred between database servers and clients. As seen, many cloud providers (e.g., Amazon EC2 [10], Rackspace [76]) bill data transferred between cloud environment and outside world. Therefore, using the CoopSC approach within a cloud-computing infrastructure presents also economic advantages. Two scenarios are considered: a) several nodes run inside a cloud environment in order to perform specific tasks which use data that originate from a database which is running outside the cloud; b) an operational database is running within a cloud environment while clients are running outside. In both scenarios, using the CoopSC approach reduces amount of data sent by database server and thus reduces amount of money that has to be paid for data transfer.

The first scenario (c.f., Figure 4.7) corresponds to non-operational use cases in which a cloud environment is used for executing specific tasks using data that originates from outside the cloud. Multiple cloud nodes are used for decreasing computation times. Clients cache and

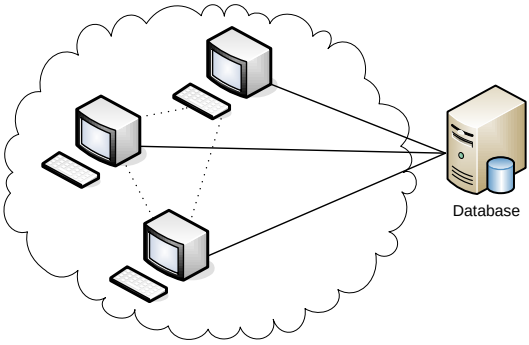

Figure 4.7: Cloud Computing Scenario A

share input data in order to reduce the communication between cloud environment and outside.

Using a cloud environment for running scientific experiments is a potential real-life use case for this scenario. These types of experiments usually involve running a number of powerful nodes inside the cloud in order to execute CPU intensive tasks [51] with data that originates from outside the cloud. In many situations, different nodes access data structures which overlap. For example, different algorithms might be executed on similar data. Applying a cooperative caching approach can have both economic and performance-wise benefits.

The second scenario (c.f., Figure 4.8) expresses operational use cases in which cloud solutions are used as alternative to constructing and maintaining a operational data center. Database server is running inside the cloud environment while clients are located outside.

For example, a corporation could use a cloud infrastructure for keeping corporate data which is accessed by clients located in geographically distributed working centers. A cooperative caching solution reduces amount of data sent by database server and, thus, decreases the monetary cost for data transfer. Since clients are exchanging tuples, using CoopSC causes an increase of data transfer at client side. Because, nowadays, most ISPs (Internet Service Providers) utilize a flat rate charging

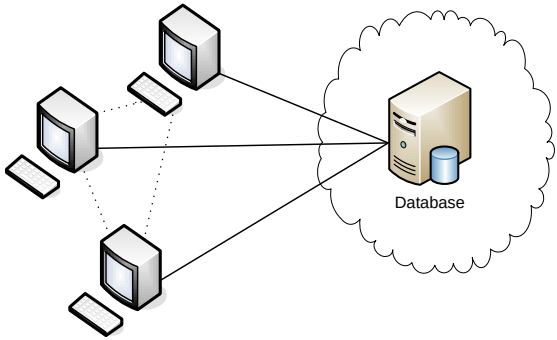

Figure 4.8: Cloud Computing Scenario B

model for bandwidth, this client-side increase of data transfer has very limited negative economic consequences.

4.6 Use Case 2: CoopSC-based Network Traffic Analysis

Analyzing IP traffic is an important task on which many network management applications are based. Cisco's NetFlow [23] system is one of the most widely used approaches for accomplishing a measurement and collection of flow records, needed to analyze traffic. NetFlow records are generated and exported by routers when communication flows end (e.g., a TCP/IP connection finishes), stored in a storing solution and accessed by analyzers in order to perform different traffic monitoring tasks (e.g., intrusion detection, accounting, charging). Every NetFlow entry has the following values: source IP address, destination IP address, IP protocol, source and destination ports (for UDP and TCP), start time, duration, number of packages and number of bytes.

Since only one records is generated per flow, NetFlow based approaches are much more scalable than solutions which monitor every IP packet. Still, working with NetFlow means to involve very large amounts of data, while the collection, storing, and enabling of analyzers demands the access to this data in the fastest possible manner. Sampling [15] and distributively storing flow records [70] emerge as important solutions

for increasing the scalability of the NetFlow collection and storage. Enabling analyzer to efficiently access stored NetFlow records is another important task which has to be performance-wise optimized.

SQL-based solutions are commonly used for storing such flow records [72]. When using such a database approach, flow records are collected and then stored in a database server and analyzers access the database and retrieve flow records in order to be used within different applications. The amount of data transferred between such analyzers and database server can be significant. Thus, reducing the traffic between the database server and analyzers is also and important objective. Range selection queries are commonly used by many flow-based analyzers. For example, many analyzers need to access flow record which were generated in a specified interval of time or which pertain to a particular subnet.

Thus, this section shows how the CoopSC architecture can be used to improve access time and data transfers between the centralized flow-based storage solution and those analyzers in operation. Such an improvement can positively influence the performance of flow-based solutions. Therefore, a new architecture NMCoopSC (Network Management CoopSC) was developed and, later evaluated throught experiments (c.f., Section 6.3).

4.6.1 Background

In order to cope with the constant increase of Internet traffic, many research projects aim to use packet and flow sampling thus, reducing the amount of traces that need to be processed [24, 27, 37]. While these approaches increase the scalability of network traffic analyzing systems, they are not very accurate in some scenarios due to the loss of information. In [15] and [64] the authors show that sampling algorithms negatively influence the performance of intrusion detection systems. The SCRIPT framework [69] uses a peer-to-peer overlay in order to store and process flow records in multiple nodes, and thus, increase the throughput of the traffic analyzing system.

In order to allow IP flows traffic accounting, Cisco introduced the NetFlow protocol for transferring IP flow records. Based on this, the IETF's (Internet Engineering Task Force) IPFIX [25] working group developed a new protocol and data format for exchanging IP flow records. The protocol was based on version 9 of Cisco NetFlow protocol. *flow_tools* [40] and *nfdump* [71] are two of the most widely used tools for storing flow records. In [72], authors show how SQL-based database relation systems can be efficiently used for storing and processing flow records.

The IETF Packet Sampling (pSAMP) working group defined a framework [36] for packet sampling a standardized different filtering and sampling techniques [97]. This framework defines which packet information are to be collected and also enables the selections of packets based on standardized mechanisms. The PSAMP protocol [26], which is based on the IPFIX protocol, allows the transfer of packet information. Based on per-packet information, characteristics related to individual packets and flows can be obtained.

Thus, existing NetFlow-based approaches aim to efficiently collect and store flow records. The newly proposed NCoopSC approach optimizes the transfer of flow-records between a logically centralized SQL-based storage solution and analyzers. A NMCoopSC-based operational solution can use all existing sampling techniques.

4.6.2 The NMCoopSC Architecture

To define the benefit of NMCoopSC to be applied to the efficient storage and handling of flow records, the NMCoopSC architecture is introduced (c.f. Figure 4.9). The flow *collector* captures and stores flow records in a SQL-based database. In order to cope with the large amount of data it involves, the collector can use sampling techniques.

The approach uses a logically centralized database storage solution. In order to improve performance, the database server might be replicated. The NMCoopSC approach is orthogonal to database replication.

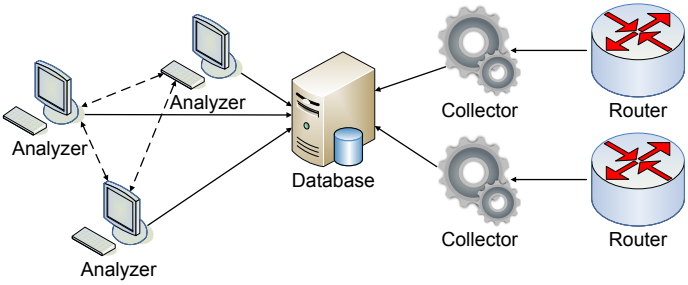

Figure 4.9: NMCoopSC Architecture

Analyzers access the NetFlow database using SQL *select* interrogations, which are assumed to be range selections. In the context of the NetFlow storage solution, range selections can be used for returning flow records which were started in a given interval of time or which pertain to a specified sub-net. Such interrogations are common in many NetFlow-based approaches. The NMCoopSC caching architecture can also be used for executing aggregation queries. This can be accomplished by creating a server side view which defines the aggregation and by using NMCoopSC with that view.

Analyzers are located in a geographically distributed environment. Such scenario can correspond to a large ISP with multiple working centers which access flow records in order to accomplish different tasks (e.g., accounting, intrusion detection systems, visualization tools). Real-life NetFlow records are also an important resource used by many research projects. The NMCoopSC approach can also be used in order to distribute NetFlow records to research centers located in different universities. Analyzers cache results of old queries and these cache entries are stored between them in a cooperative way.

4.7 Chapter Summary

Based on the general requirements of the CoopSC approach, which were described in Chapter 3, this chapter presented the architecture of the

newly CoopSC system which implements the general cooperative semantic caching approach. The main components of this architecture and their interactions were identified and described. Since updates are handled using the database server, the database-side update related CoopSC components were also presented. Moreover, the design of the protocols used during communication was outlined. Finally, this chapter outlines also the architecture of the two CoopSC-based real-life motivating use-cases which are developed during this cases (c.f. Sections 4.5 and 4.6). The CoopSC architecture was implemented (c.f., Chapter 5) and evaluated with positive performance-wise results (c.f., Chapter 6).

5
System Implementation

This chapter describes the implementation of the newly developed CoopSC system which fully implements the architecture outlined in the previous chapter. The CoopSC prototype allows database applications to take advantage of the proposed cooperative database semantic caching approach, and supports the PostgreSQL [67] system, which is one of the most popular and performant open-source database management systems.

Since decreasing query response time is one of the main goals of CoopSC, maximizing the performance has been the most important factor which was taken into consideration when designing and implementing the CoopSC system. The ease of the development process was another factor which was considered when choosing the development environment.

Taken into account these two important factors, C++ [85] was chosen as the programming language in which the CoopSC project was developed. C++ compilers generate efficient machine codes which are executed directly by the CPU [53]. Higher level languages (such as Java, C# or Python) generate *bytecode* which is then executed by virtual machine. This extra layer of execution introduces a performance penalty. The C++

language also supports the object oriented programming paradigms, and thus improves the general development process. The *Boost* library [58] collection was extensively used during implementation. This collection provides many useful object-oriented libraries for working with threads, smart pointers, and parsers. The *lippq* library [67] is used for the communication between the CoopSC middleware and PostgreSQL.

While not aiming to provide a full development documentation of the project, in what follows, this chapter describes the most important development components of the CoopSC system, arguing the most important design decisions. Thus, initially the *query plan tree* is presented in Section 5.1. The *storage module* is described in Section 5.2. Furthermore, details about the *database* and *peer executors* are outlined in Sections 5.3 and 5.4. Next, the *distributed index* (c.f., Section 5.5) and the *CoopSC metadata* component (c.f., Section 5.6) are presented. Section 5.7 presents the *CoopSC API* while the *CoopSC GUI* is described in Section 5.8. Details about the implemenation and the deployment of the two use cases which are analyzed during this thesis are outline in Sections 5.9 and 5.10 Finally, this chapter is summarized in Section 5.11.

5.1 Query Plan Tree

A *query plan tree*, which is returned by the query rewriting process, describes how a query is to be executed using local cache, remote caches or the database server. Its implementation is done using the classic *composite* design pattern [75]: a *node* super-class contains a list of children. Each operation is implemented in a class which inherits the *node* class.

Figure 5.1 presents the UML diagram of the classes which are used for constructing query plan trees. The main *QueryPlanTree* class contains a *root* node and a method for executing the query plan tree (*QueryPlanTree::execute*). This method has, as argument, the timestamps of semantic regions which intersect given query. These timestamps are used in order to make sure that query execution does not use old semantic regions.

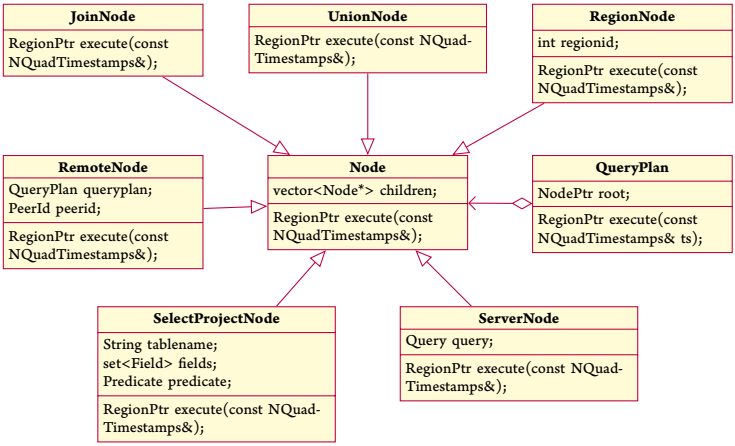

Figure 5.1: Query Plan Tree UML Diagram

The *Node* abstract super-class contains a list of children and an abstract *execute* method. Each type of operation is implemented as a subclass of the *Node* class. Thus, *Union*, *Join*, *SelectProject* implement the basic union, join and select-project operations. *RemoteNode* executes a query plan tree on a different remote client. *RegionNode* accesses a local semantic region, while *ServerNode* sends a query for execution to the database server.

5.2 Storage Module

The storage module handles the storage of semantic regions. Its implementation must, thus, contain a class for expressing the concept of semantic regions and also a different class must be designed for handling the management of these regions.

Figure 5.2 presents the UML diagram of the classes which implement the storage functionality inside the CoopSC system. The *Region* class stores the content of semantic regions which were generated as results of old query executions. An instance of this class has a query description (*Region::query*), a set of tuples which form the current semantic region

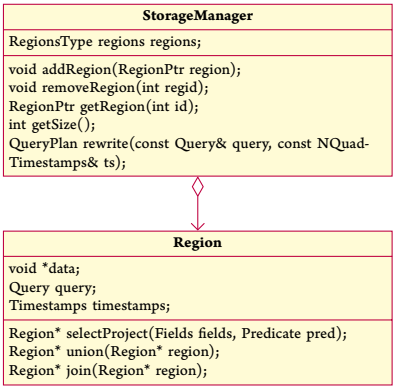

Figure 5.2: Storage UML Diagram

(*Region::data*) and timestamps which were returned from the database server when this semantic region was generated (*Region::timestamps*). Methods are provided for executing basic database operations (*select-project*, *union* and *join*).

The *StorageManager* class manages the storage of semantic regions. A *StorageManager* instance keeps a list of semantic region and provides methods for adding (*StorageManager::addRegion*), removing (*StorageManager::removeRegion*), and retrieving (*StorageManager::getRegion*) such regions. When adding or removing entries from/to the local cache, the *distributed index* is also updated. The LRU (Least Recently Used) replacement policy is also implemented in this class. Furthermore, this class implements the local rewriting algorithm (*StorageManager::rewrite*).

5.3 Database Executor

The *database executor* (c.f., Figure 5.3) handles executions of sub-queries on the database server. In order to allow multiple queries to be executed in parallel on the server, a connection pool mechanism is used. Thus,

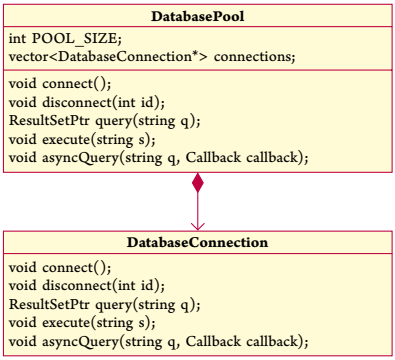

Figure 5.3: Database Executor UML Class Diagram

the class *DatabasePool* contains a list of database connections. The size of this pool is statically specified. When executing a query, a random connection is chosen from the pool and the given query request is redirected to the associated *DatabaseConnection* object.

Methods are provided for executing queries (*DatabaseConnection::query*) and statements (*DatabaseConnection::execute*). An asynchronous queries execution facility is also provided (*DatabaseConnection::asyncQuery*). This functionality is used during the execution of a *query plan tree* in order to parallelize executions of different nodes.

5.4 Peer Executor

The *Peer Executor* component (c.f., Figure 5.4) handles executions of *remote probe* sub-queries on remote peers. Similarly with the *database executor* component, a connection pool approach is implemented in order to reuse single connections to remote peers for multiple query executions. Methods are provided or executing subqueries both in synchronous (*Peer::execute*) and asynchronous matter (*Peer::asyncExecute*).

A communication protocol was designed (c.f., Section 4.4) for sending requests and returning results of executions. This protocol works on

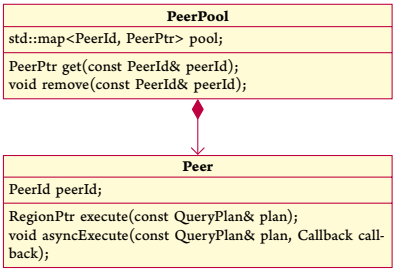

Figure 5.4: Peer Executor UML Class Diagram

top of the TCP/IP transport protocol and uses the *boost::serialization* library [58] for serializing request/response objects.

5.5 Distributed Index

The *distributed index* is used for indexing semantic regions in order to be available to all clients. The approach described in Section 3.3, was implemented on top of P2P overlay network which is formed by all clients which are interrogating a specific database server.

Figure 5.5 presents the UML diagram of the distributed index component. The *DistributedIndex* class contains methods for adding (*DistributedIndex::add*) and removing (*DistributedIndex::remove*) semantic regions to the distributed index. The distributed rewriting algorithm is also implemented in this class (*DistributedIndex::rewrite*). In order to make sure that the rewriting process does not use old entries, the *rewrite* method has, as an argument, a list of virtual timestamps of quads which intersect given query.

The distributed index is built on top of a P2P overlay which implements a KBR (key-based routing) algorithm. The CoopSC system uses the Chimera P2P system [22]. Chimera was chosen because it is a lightweight and efficient P2P overlay developed in the C programming language which can be easily integrated in a C++ project.

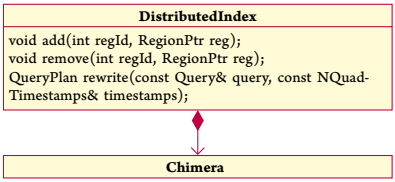

Figure 5.5: Distributed Index UML Class Diagram

```
CREATE TABLE coopsc.metadata
(
  id serial NOT NULL,
  table_name character varying(50),
  fields character varying(100),
  minlevel integer,
  maxlevel integer,
  tslevel integer,
  CONSTRAINT coopsc_metadata_pkey PRIMARY KEY (id )
)
```

Figure 5.6: CoopSC Metadata

5.6 CoopSC Metadata

The *CoopSC Metadata* contains data about tables and fields for which the CoopSC approach was enabled and also keeps some numerical parameters related to the configuration of the distributed index and the update handling mechanism. Since the metadata must be accesable to all clients during the initialization phase, it is stored on the database server side as a relational table.

Figure 5.6 illustrates the structure of the CoopSC metadata relation. *table_name* specifies the names of tables for which the CoopSC approach is enabled. The names of fields are specified, as a comma separated list, in the column *fields* (e.g., *"latitude, longitude"*). *minlevel* and *maxlevel* indicate the minimum and the maximum level of the MX-CIF quad tree-based distributed index, while *tslevel* (timestamps level) specifies the level at which timestamps are computed in order to handle update statements.

5.7 CoopSC API

In order to allow users to develop database applications that take advantage of the cooperative semantic caching approach, the CoopSC system must provide an API (Application Programming Interface). Such an API must offer ways of creating CoopSC connections, executing queries, and returning their results sets. These result sets must be also accessible to clients in an iterative way.

The CoopSC API consists of two main classes: *coopsc::Connection* (c.f., Figure 5.7) and *coopsc::ResultSet* (c.f., Figure 5.8). The *coopsc::Connection* provides methods for establishing the CoopSC environment and executing queries and statements. The configuration parameters of the CoopSC are specified in the constructor. Database name (*dbname*), host (*dbhost*), port (*dbport*), user (*dbusr*), and password (*dbpwd*) configure connection to the PostgreSQL database server. CoopSC host (*coopschost*) and port (*coopscport*) specify local address which is used for CoopSC connections and port on which system listens for requests from other peers. The next arguments configure the type (*cachetype*) and size (*cachesize*) of the cooperative semantic cache. The last parameters of the constructor configure the Chimera P2P overlay connection (local host, local port, bootstrap host, bootstrap port).

The method *Connection::query* executes a query using the cooperative cache, while *Connection::queryServer* bypasses the cache by executing a query directly on server. These two methods return *coopsc::ResultSet* object instances. *Connection::execute* executes a non-select statements (e.g., *insert, update*).

The class *coopsc::ResultSet* provides methods for accessing result set returned by CoopSC queries. *ResultSet::getNoTuples* and *ResultSet::getNoFields* return the total number of tuples and fields contained in current result set. Names of fields are returned by the *ResultSet::getFieldName* method. The methods *Result::getValue* are used for

```
typedef enum
{
    NoCache = 0, LocalCache, CooperativeCache
} CacheType;

class Connection
{
public:
    Connection(
                //Database connection configuration
                const std::string& dbname,
                const std::string& dbhost,
                int dbport,
                const std::string& dbusr,
                const std::string& dbpwd,

                //CoopSC configuration
                const std::string& coopschost,
                int coopscport,

                //Cache configuration
                CacheType cachetype = CooperativeCache,
                unsigned int cachesize = 128 * 1024 * 1024,

                //Chimera configuration
                const std::string& chimerahost = "",
                int chimeraport = 9999,
                const std::string& chimerabshost = "",
                int chimerabsport = -1);

    void connect();
    void disconnect();

    //executes a query using CoopSC
    ResultSetPtr query(const std::string& q);
    //executes an SQL statement
    void execute(const std::string& s);

    //executes a query directly on seimplapirver
    ResultSetPtr queryServer(const std::string& q);
};
```

Figure 5.7: CoopSC API

iterating through tuples contained in result sets. Finally, there are also methods provided for determining the type and size of these fields.

Figure 5.9 presents a simple example which uses the CoopSC API in order to execute a simple one-dimensional selection query. Ini-

```
class ResultSet
{
public:
    unsigned int getNoTuples();
    unsigned int getNoFields();
    std::string getFieldName(int fieldno);

    std::string getValue(unsigned int tupleno,
                         unsigned int fieldno);
    std::string getValue(unsigned int tupleno,
                         const std::string& field);

    std::string getFieldType(int fieldno);
    std::string getFieldType(const std::string& field);

    int getFieldSize(int fieldno);
    int getFieldSize(const std::string& field);
};
```

Figure 5.8: CoopSC Result Set

tially, a CoopSC connection object is created by specifying all configuration parameters related to PostgreSQL, CoopSC, cache type and size, and Chimera. The method *Connection::connect* is used for establishing connection. Finally, the query string is executed using the *Connection::query* method. It should be noted that CoopSC connections are destroyed by the destructor of the CoopSC class in a typical RAII (Resource Acquisition Is Initialization) matter [65] and thus, calling *Connection::disconnect* is not necessary.

5.7.1 JDBC Driver

While the CoopSC API allows developing database application which take advantage of the cooperative semantic caching, adapting existing projects in order to use the CoopSC system can be difficult since it would involve changing all code that handles database access. Thus, mechanisms must also be provided for allowing existing database applications to easily be adapted for using the CoopSC system.

```
#include <iostream>
#include <string>
#include <coopsc.h>

using namespace std;
using namespace boost;

int main(int argc, char **argv)
{
  coopsc::Connection conn("wisconsin", "192.41.135.205",
                          5143, "test", "test",

                          "192.168.0.2implapi", 10002,
                           coopsc::CooperativeCache, 1 << 25,

                          "192.168.0.2", 4002,
                          "192.168.0.1", 4001);
  conn.connect();
  string sql;
  sql = "select * from wisconsin where "
        "0 < unique1 and unique1 < 10000";

  ResultSetPtr result = conn.query(sql);
}
```

Figure 5.9: CoopSC Example

JDBC (Java Database Connectivity) [77] is a Java-based technology which enable applications to access dimplapiifferent types of database systems in a standardized way. Database providers can be changed with minimum modifications performed on application code.

A CoopSC JDBC driver was implemented in order to permit existing applications to be easily adapted in order to use the CoopSC system. The implementation uses the JNI (Java Native Interface) technology [45] for bridging the Java and C++ environments. Each JDBC call is mapped to the corresponding CoopSC API method by the driver and thus, applications can use the cooperative semantic caching in a transparent matter.

Figure 5.10 illustrates how the CoopSC system is used within a JDBC-based database application. The PostgreSQL database host, port and name are specified in a JDBC connection URL, while the CoopSC-related configuration parameters are stored in the *coopSCInfo* property

```
public class CoopSCTest {

 private static Connection getConnection()
        throws SQLException, ClassNotFoundException {
   Class.forName("ch.uzh.coopscjdbc.CoopSCDriver");

   Properties coopSCInfo = new Properties()
   String url;
   url = "jdbc:coopsc://192.41.135.20implapi5:5432/wisconsin";
   coopSCInfo.setProperty("user", "test");
   coopSCInfo.setProperty("password", "test");
   coopSCInfo.setProperty("cachetype", "COOPERATIVE_CACHE");
   coopSCInfo.setProperty("cachesize", "33554432");
   coopSCInfo.setProperty("coopschost", "192.168.0.2");
   coopSCInfo.setProperty("coopscport", "10002");
   coopSCInfo.setProperty("chimerahost", "192.168.0.2");
   coopSCInfo.setProperty("chimeraport", "4002");
   coopSCInfo.setProperty("chimerabshost", "192.168.0.1");
   coopSCInfo.setProperty("chimerabsport", "4001");
   return DriverManager.getConnection(url, coopSCInfo);
 }

 public static void main(String[] args) throws SQLException {
   Connection con = getConnection();
   Statement stmt = con.createStatement();
   String sql;
   sql = "select␣*␣from␣wisconsin␣where␣" +
         "o␣<␣unique1␣and␣unique1␣<␣10000";
   stmt.execute(sql);
   ResultSet result = stmt.getResultSet()

   stmt.close();
   conn.close();

 }
}
```

Figure 5.10: CoopSC JDBC Example

map. Both the URL and the property map specified when creating the JDBC CoopSC connection object.

5.8 Graphical User Interface

In order to demonstrate the functionality of the CoopSC architecture, a graphical user interface (GUI) was also implemented. The

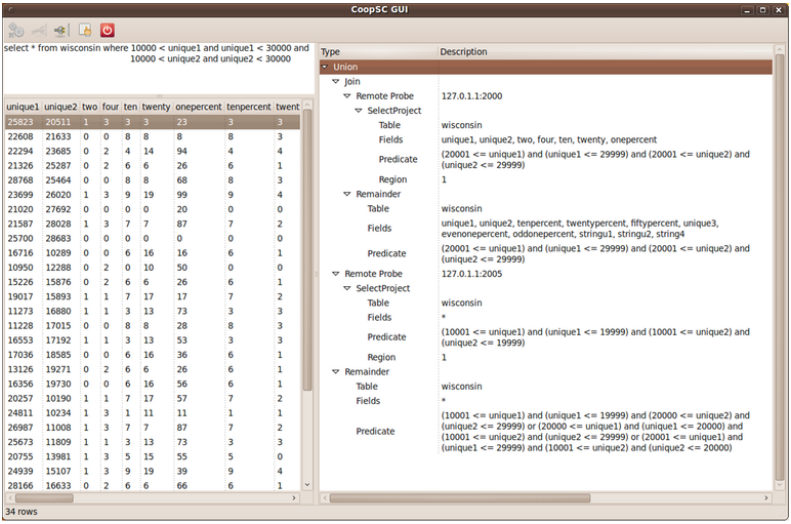

Figure 5.11: CoopSC GUI

CoopSC GUI enables users to execute SQL (Structured Query Language) queries using the CoopSC cooperative cache. Both results of query executions and the way in which queries were executed are displayed.

Figure 5.11 illustrates the CoopSC graphical user interface. The application window is divided into three parts. The upper-left text box allows user to enter queries that are to be executed using CoopSC. The lower-left table displays results of query executions. The right side of the window contains a tree structure that shows how queries were split into probe, remote probes and remainders.

5.9 Use Case 1: CoopSC within Cloud Environments

This use-case studies both the economic- and performance-wise advantages of using CoopSC within cloud computing environments (c.f., Section 4.5). Two scenarios are considered: a) several nodes run inside a cloud environment while the database is running outside the cloud; b)

database is running within a cloud environment while clients are running outside.

Based on the general implementation of the CoopSC approach, these two scenarios are deployed by running a PostgreSQL database and multiple client nodes which are using the CoopSC system. Clients use the CoopSC API (c.f., Section 5.7) in order to execute SQL interrogations.

The second scenario assumes that a relational database server is running inside a cloud environment. While the current CoopSC implementation works only with the PostgreSQL database server, which it is assumed to be running inside a cloud-based virtual machine (e.g., Amazon EC2, Rackspace), the general CoopSC approach can be easily adapted in order to work with relation SQL-based cloud services such as Microsoft SQL Azure [66] or Amazon RDS (Relational Database Service) [3].

5.10 Use Case 2: CoopSC-based Network Traffic Analysis

The NMCoopSC use-case (c.f., Section 4.6) utilizes the CoopSC approach is order to improve the performance of NetFlow-based traffic analysis. NetFlow entries are stored in a logically centralized database which is accessed by CoopSC-enabled clients located in a geographically distributed environment.

The NetFlow database schema is illustrated in Figure 5.12. Each NetFlow record corresponds to a tuple from the *flows* table. In order to improve performance, indexes are also created for columns involved in range interrogations. *Starttime* is represented in the Unix time format (i.e. the number of seconds elapsed since January 1, 1970). *Duration* is expressed in seconds. *Proto* specifies the protocol (e.g., TCP, UDP, ICMP). *srcip* and *srcport* specify the source ip and port of the flow. *dstip* and *dstport* identify the destination. *packets* represents the number of packets contained the flow, while *bytes* the amount of bytes transferred.

```
CREATE TABLE flows
 (
  id integer, starttime integer,
  duration integer, proto integer,
  srcip integer, srcport integer,
  dstip integer, dstport integer,
  packets integer, bytes integer,
  CONSTRAINT flowspkey PRIMARY KEY (id)
 )
```

Figure 5.12: NMCoopSC Database Schema

5.11 Chapter Summary

This chapter gives an overview of the implementation of the CoopSC system, which was based on the general CoopSC architecture (c.f., Chapter 4). The most important implementation modules were described, presenting their UML class diagrams. A *query plan tree* determines how a query is to be executed using cooperative cache and its implementation is using the *composite design pattern*. The *storage module* stores semantic regions, manages the local cache and it is also used from executing the local rewriting process. *Database* and *peer executors* execute sub-queries on database servers or remote peers and are both implemented using a connection pool mechanism. The *distributed index*, which is used for indexing semantic regions, is using the Chimera P2P overlay. Also, the API which allows database applications to use the CoopSC system was presented, documented and exemplified.

6
Evaluation

In order to show the benefits of the cooperative semantic caching approach, the newly developed CoopSC system was extensively evaluated through experiments. These experiments show how the performance of client-server database management systems is influenced by the CoopSC, by comparing the cooperative semantic approach with the classic semantic caching solution and to the default no-caching database client-server environment. The behavior of the CoopSC system is also evaluated by looking at the hit rates, tuples' origins, and duration of distributed rewriting. The economic benefits of using CoopSC within cloud environments were investigated by applying and experimenting with CoopSC within real-life cloud infrastructures. Finally, the NM-CoopSC architecture, which was based on CoopSC, was evaluated with real life NetFlow data.

In order to emulate real-life production CoopSC use cases, the experiments were performed using the EMANICSLab [39] distributed testing infrastructure. This infrastructure allows creating virtualized nodes within a European-wide distributed network.

Thus, this chapter is organized as follows: initially the result of the general CoopSC evaluation is presented (Section 6.1). The general eval-

Table 6.1: EMANICSLab Hosts

Id	Host
1	emanicslab1.csg.uzh.ch
2	emanicslab2.csg.uzh.ch
3	emanicslab1.informatik.unibw-muenchen.de
4	emanicslab2.informatik.unibw-muenchen.de
5	emanicslab1.ewi.utwente.nl
6	emanicslab2.ewi.utwente.nl
7	host1-plb.loria.fr
8	host2-plb.loria.fr
9	emanics2.ee.ucl.ac.uk
10	emanicslab1.eecs.jacobs-university.de

uation uses the Wisconsin benchmark [14] synthetic data set. This is followed by the evaluation of the two CoopSC cloud scenarios, described in Chapter 4.5 (Section 6.2). The NMCoopSC architecture is then evaluated using real-life NetFlow data (Section 6.3). Finally, the results of these evaluations are summarized and some concluding remarks are made in Section 6.4.

6.1 CoopSC System

The CoopSC was evaluated using a PostgreSQL database server and a of clients that execute, in parallel, single and double indexed attribute selection queries. Updates statements were also evaluated. During the evaluation, three scenarios were used: cooperative semantic caching approach, classic semantic caching and no caching approach.

The EMANICSLab research testing network was used for this evaluation. Clients are running in 10 nodes located across Europe, while the database server runs on a more powerful machine located in Zurich. Table 6.1 contains the list of EMANICSLab nodes which were used for this evaluation, while Table 6.2 presents the round-trip times (RTT) between each peer of nodes, expressed in milliseconds. Table 6.3 contains

Table 6.2: EMANICSLab RTT

RTT	1	2	3	4	5	6	7	8	9	10
1		0.19	34.44	34.45	26.34	26.34	18.08	17.98	21.17	38.82
2			34.78	34.53	26.36	26.38	18.13	18.00	21.24	38.91
3				0.30	19.80	19.45	28.86	26.59	30.97	17.01
4					19.04	18.89	28.41	26.93	29.91	16.46
5						0.20	25.07	24.82	11.53	17.01
6							25.17	24.91	11.55	16.97
7								0.37	13.66	28.08
8									13.57	27.98
9										34.41
10										

Table 6.3: EMANICSLab Throughput

Bit rate	1	2	3	4	5	6	7	8	9	10
1		94.3	13.3	14.2	18.0	17.9	24.9	25.0	21.8	12.4
2	94.3		14.2	14.2	18.0	18.0	24.8	25.0	21.8	12.4
3	13.7	13.2		622	24.6	25.5	16.4	17.8	15.0	27.0
4	14.2	13.0	632		25.7	25.5	16.5	17.8	16.1	29.1
5	9.68	9.78	11.6	11.1		828	7.18	8.34	17.5	19.6
6	7.36	6.38	11.4	11.8	933		8.35	7.73	18.2	11.9
7	24.8	23.5	17.1	17.0	18.4	18.6		94.3	31.5	16.6
8	25.0	23.7	15.8	14.2	18.7	18.7	94.3		31.9	16.6
9	21.6	20.8	15.4	15.9	36.5	36.5	31.5	31.7		13.4
10	12.3	12.3	24.3	28.7	28.0	27.8	16.5	16.7	13.7	

the throughput, in Mbits/sec, of sending data from the host associated with to row to the host determined by the column.

The evaluation was done using the Wisconsin benchmark [14] relation of 10 million tuples, where each tuple contains 208 bytes of data. Each query is a range selection on either the *unique1* attribute (Example: select * from wisconsin where 4813305 < unique1 and unique1 < 4823306) or on *unique1* and *unique2* (Example: select * from wisconsin where 4813305 < unique1 and unique1 < 4823306 and 23000 < unique1 and unique1 < 33000).

Similarly with the evaluation of other cache architectures [5], [6], queries executed by each client have a semantic locality. For each client, the center-points of queries were randomly chosen to follow a normal

distribution curve with a chosen standard deviation. For each experiment, clients first execute warm-up queries until cache is filled.

The response time, for each client, is calculated by averaging the response time of 10 testing sessions of 50 queries each. The error bar is calculated using these 10 values. For each scenario, the total amount of data sent by database server is also measured. Furthermore, for the cooperative caching scenario, in each session, numbers of tuples that originated from the local cache, remote caches and the database server are determined. *Local* and *peers* hit rates are computed for the cooperative caching scenario. The local hit rate is defined as the percent of tuples from result sets that originated from the local cache, while peers hit rate refers to tuples that were returned from caches of the other clients. The duration of the distributed rewriting process is also measured.

Thus, in each experiment, five measurements are made: the first two compare the cooperative semantic caching approach with classic semantic caching and no caching scenario, in the relation to (a) query response time and (b) amount of data sent by database server. The other three measurements refer only to the cooperative caching approach and determine (c) tuples' origin, (d) hit rates and (e) duration of the distributed rewriting.

For single attribute selections experiments, the distributed index was configured with the following parameters $f_{min} = 10, f_{max} = 18, f_{update} = 15$. For the double attributes workload these parameters were $f_{min} = 15, f_{max} = 20, f_{update} = 18$. These values were chosen in order to determine a good distribution of the values stored in the distributed index and using the Wisconsin benchmark dataset.

6.1.1 Cache Size

The first experiment measures how the variation of the size of clients' caches influences the performance of the two caching approaches for single-attribute selections. The size of clients' caches are varied from 0 to 192 MB. This experiment uses 10 clients, which is a reasonable values for the CoopSC real-life use cases. These workloads have standard de-

viations of 500,000. The means of the Gaussian curves are distributed uniformly over the range of the *unique1* attribute. The difference between the means of two consecutive clients is 300,000. This values were chosen in order to determine a workload in which queries has a moderate semantic locality which corresponds to many real-life use cases. The evaluation was stopped at 192 MB when that tendency of the system behaviour became clear.

Analyzing the response time (Figure 6.1), for small cache sizes, the difference between the two approaches is reduced, because hit rates are small in both scenarios and the database server has to handle executions of most queries. While the cache sizes increase, the benefits of the cooperative caching approach become more visible. For large cache sizes, the difference becomes again reduced, because a large part of queries can be answered completely by accessing only the local cache and, thus, in many situation the cooperative cache is not needed. In the semantic caching approach, the number of tuples sent by the database server is reduced (Figure 6.2), because the database server only sends parts of queries which are missing from the local cache. The cooperative approach further decreases the number of tuples sent because clients can also transfer tuples from caches of other peers.

Figure 6.3 illustrates the origin of tuples for the cooperative caching approach. For small cache size, most tuples are returned from the database server. As cache size increases, both the number of tuples returned from the local cache and caches of other clients increase. For larger cache size, number of tuples returned from caches of other peers decreases because most queries can be answered using entries from the local cache and thus, cooperation is reduced.

Hit rates have a similar behavior (Figure 6.4). For small cache size both the local and peers hit rate are reduced. Increasing cache size causes the increase of both hit rates. The peers hit rate reaches a maximum, and then starts to decrease because the need of cooperation decreases.

The duration of the distributed rewriting process (Figure 6.5) increases, while the cache sizes grow, because the number of semantic regions stored in the distributed index also becomes higher. For larger

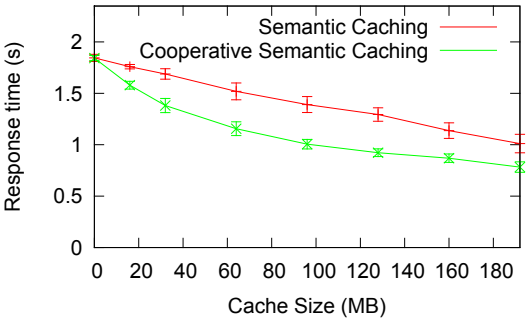

Figure 6.1: Cache Size: Response Time

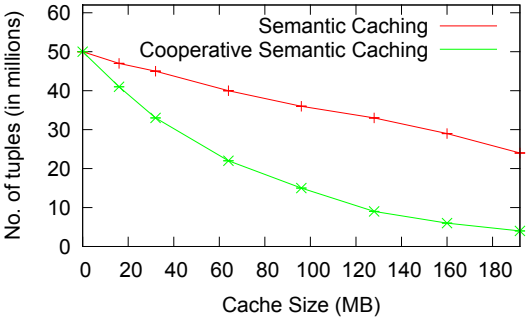

Figure 6.2: Cache Size: Server Tuples

cache sizes, this duration starts to decrease because most queries are answered using only local caches and thus, smaller sub-queries are sent for rewriting to the distributed index.

6.1.2 Cache Size (2-dimensional Scenario)

A similar experiment was executed for two dimensional selections. The workloads have standard deviations of 1,000,000 on both attributes. Queries are rectangles with side lengths of 300,000 and return around 9,000 tuples.

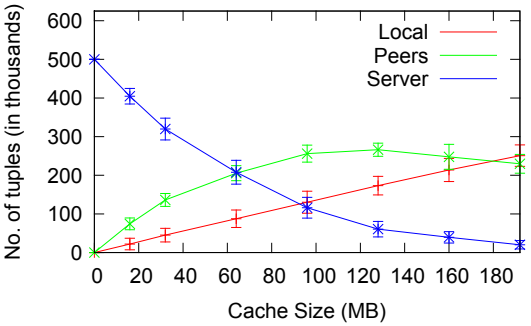

Figure 6.3: Cache Size: Tuples' Origin

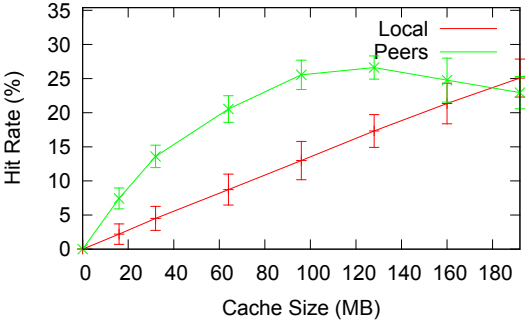

Figure 6.4: Cache Size: Hit Rate

The results are illustrated in Figures 6.6, 6.7, 6.8, 6.9, and 6.10. Except response time (Figure 6.6), the other measurements are similar with the one from the single attribute selection. For two dimensional selections, the time-wise cost of query rewriting and accessing the distributed indexed is increased. For small cache size, due to the low peers hit rate, this cost overcomes the benefits of cooperation and thus, the cooperative caching approach performs worst than semantic caching. With the increase of cache size, the cooperative approach starts to outperform semantic caching because the hit rates increase which compensates for the cost of query rewriting.

Compared with the one-dimensional selections experiment, the duration of the distributed rewriting (Figures 6.5 and 6.10) exhibits a similar behavior but the absolute values are higher because the complexity

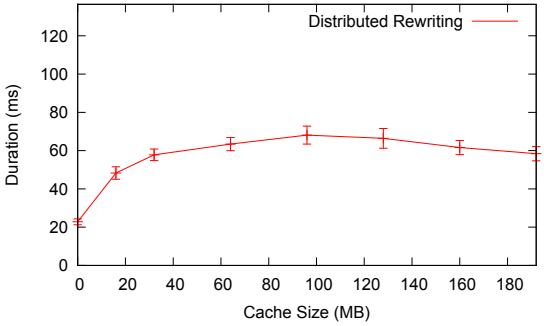

Figure 6.5: Cache Size: Rewriting Duration

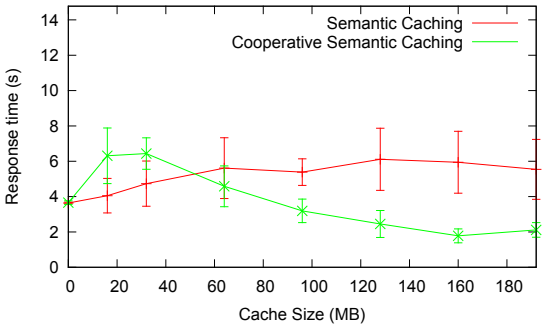

Figure 6.6: Cache Size (2-dimensional Scenario): Response Time

of the distributed index grows exponentially with the number of dimensions.

6.1.3 Locality

The next experiment measures how varying queries' locality influences the performance of both caching approaches. The size of clients' cache is 64 MB. We chose this value because the workloads generated for this evaluation exhibit a moderate behavior for this cache size. The experi-

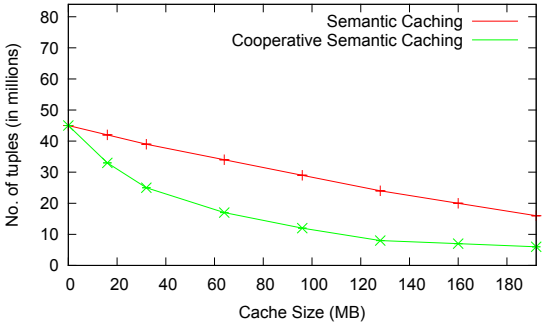

Figure 6.7: Cache Size (2-dimensional Scenario): Server Tuples

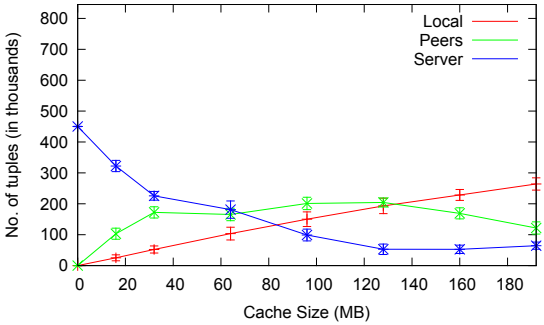

Figure 6.8: Cache Size (2-dimensional Scenario): Tuples' Origin

ment uses 10 clients. The workloads' standard deviations are varied from 100,000 to 1,000,000.

Increasing standard deviation of workloads decreases access locality and thus, the performance of both caching systems decreases (Figures 6.11 and 6.12).

In the cooperative caching approach, increasing the standard deviation causes the number of tuples returned from the local cache to decrease, while the number of tuples returned from the server increases. The number of tuples returned from remote peers initially increases, because lowering access locality increases the need to cooperate. A further increase of standard deviation decreases also the amount of data returned from remote peers because with a general low access locality, relevant entries will not be found in other peers (Figure 6.13).

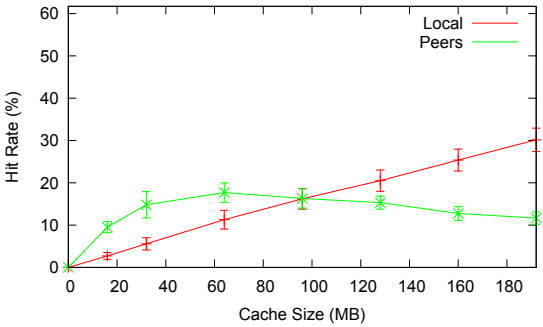

Figure 6.9: Cache Size (2-dimensional Scenario): Hit Rate

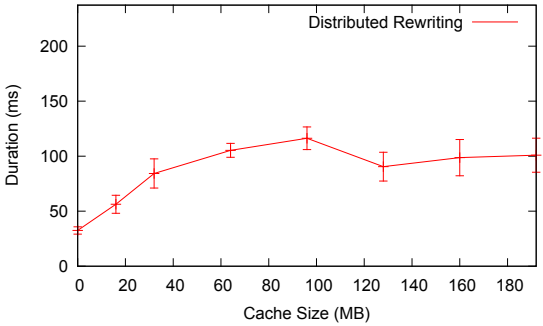

Figure 6.10: Cache Size (2-dimensional Scenario): Rewriting Duration

The local hit rate decreases with the increase of the standard deviation while peers hit rate initially increases and then decreases (Figure 6.14).

For higher locality workloads the distributed rewriting is faster since larger parts of queries are answered using only local caches and thus, smaller sub-queries are sent for rewriting to the distributed index (Figure 6.15). While deviation increases, the duration of distributed rewriting follows a similar trend.

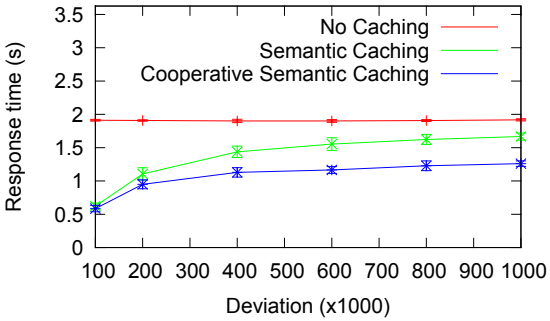

Figure 6.11: Deviation (2-dimensional Scenario): Response Time

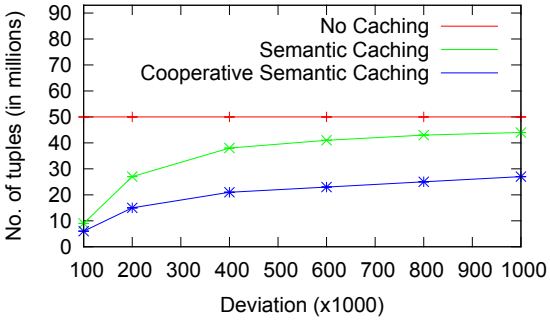

Figure 6.12: Deviation (2-dimensional Scenario): Server Tuples

6.1.4 Locality (2-dimensional Scenario)

A similar locality experiment was performed for 2-dimensional selection queries. In this experiment, the locality of both *unique1* and *unique2* attributes are varied from 100,000 to 1,000,000.

Similarly to the evaluation of the one-dimensional workload, increasing standard deviation of workloads decreases access locality and thus, the performance of both caching systems decreases (Figures 6.16 and 6.17).

The hit rate and tuples' origins measurements (Figures 6.19 and 6.18) exhibit also a similar behaviour with the previous experiment.

The rewriting duration (Figure 6.20) first increases because less queries are answered using local cache, and thus, more queries are sent

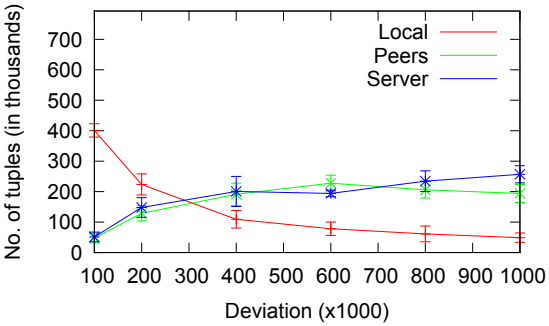

Figure 6.13: Deviation (2-dimensional Scenario): Tuples' Origin

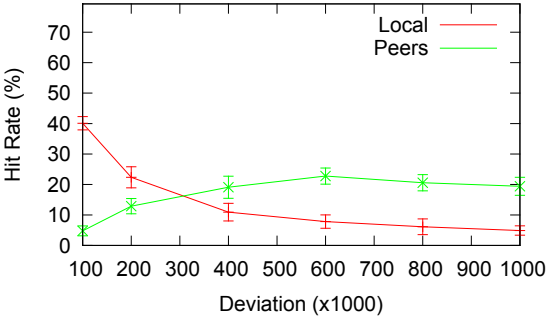

Figure 6.14: Deviation (2-dimensional Scenario): Hit Rate

for distributed rewriting. For larger deviations, this duration begins to decrease because most queries are not answered using the cooperative cache and thus, the rewriting process uses less time to return fewer results.

6.1.5 Query Size

The next experiment measures how the size of selections influences the performance of the system. The size of selections on the *unique1* at-

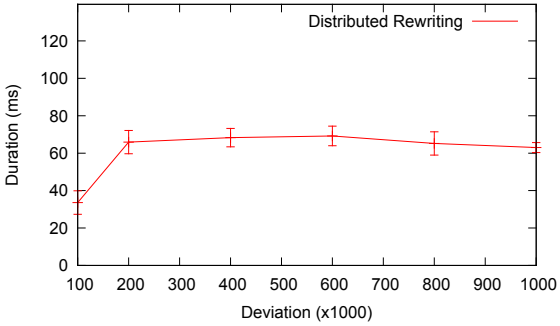

Figure 6.15: Deviation: Hit Rate

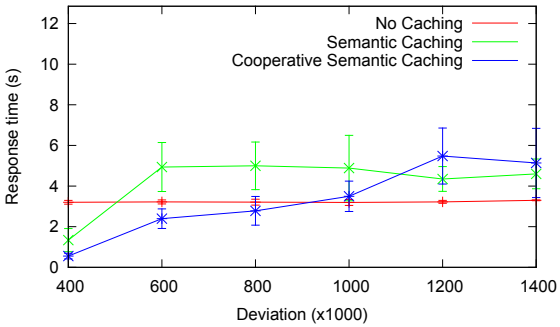

Figure 6.16: Deviation (2-dimensional Scenario): Response Time

tribute is varied from 1,000 to 20,000 tuples. This interval provides a good variation from small to large query result sizes.

As the size of selections increases, the response times (Figure 6.21) and the amount of data sent by database server (Figure 6.22) also increase. The semantic caching approach is more efficient than the no-caching approach because database server only handles parts of queries which can not be answered using the cache. The cooperative caching solution outperforms the local caching approach due to the increase of hit rate of the cache system.

The hit rate remains constant since the same proportions of queries are answered using cache local and remote caches (Figure 6.24).

Increasing the size of selections causes a higher number of tuples to be returned by individual queries. Thus, both the number of tuples re-

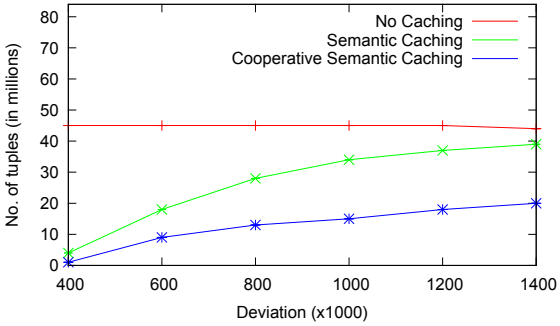

Figure 6.17: Deviation (2-dimensional Scenario): Server Tuples

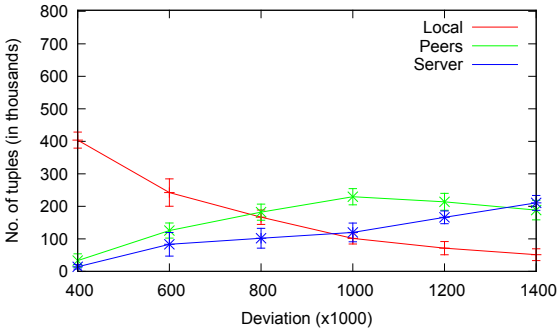

Figure 6.18: Deviation (2-dimensional Scenario): Tuples' Origin

turns from caches (local or remote) and the database server increase (Figure 6.23).

The duration of the distributed rewriting slightly decreases with the increase of selection size, because larger queries are indexed at lower levels and thus the communication overhead decreases (Figure 6.25).

6.1.6 Number of Clients

Evaluations were also performed in order to determine how the performance of the system varies while increasing the number of clients. The

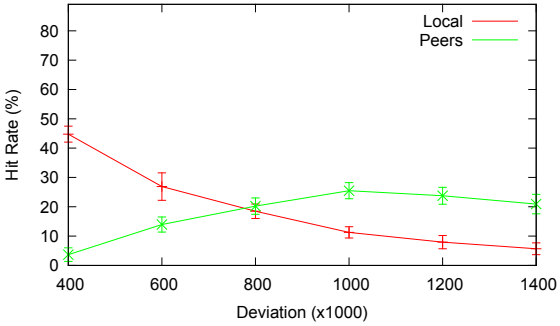

Figure 6.19: Deviation (2-dimensional Scenario): Hit Rate

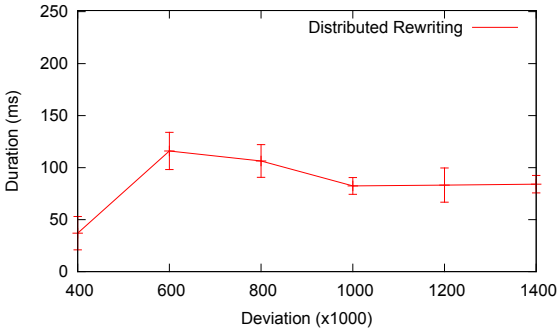

Figure 6.20: Deviation (2-dimensional Scenario): Hit Rate

number of clients are varied from 1 to 30. When 30 clients are used the response time for the no-caching scenario become unreasonable high (more then 6 seconds) and the evaluation is stopped. As it can be seen, both the response time and the amount of data sent by server when using the cooperative caching approach are lower compared to the scenario when no caching is used or when only local semantic caching is utilized (Figure 6.26 and 6.27). As the number of clients increases, the database server has to handle the execution of more queries in parallel, thus, the average response time and the amount of data sent increase. When running queries using the semantic caching approach, the server has to execute only parts of these queries that were not found in local caches of these clients. This decreases the average response time and the amount of data sent. When using CoopSC, cache entries are shared between

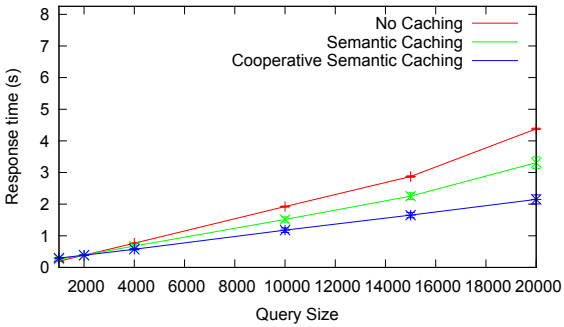

Figure 6.21: Query Size: Response Time

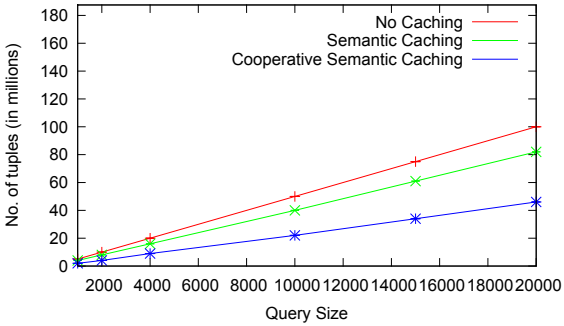

Figure 6.22: Query Size: Server Tuples

clients. This causes a further decrease of the average response time and of the amount of data sent, because the hit rate of the cache system increases.

Except for the single client scenario, hit rates remain constant since the same portion of queries are answered using the local and remote caches (Figure 6.29). When this experiment runs with a single the remote hit rate is obviously 0. Tuples' origin inhibit a similar behavior (Figure 6.28). The distributed rewriting durations (Figure 6.30) in-

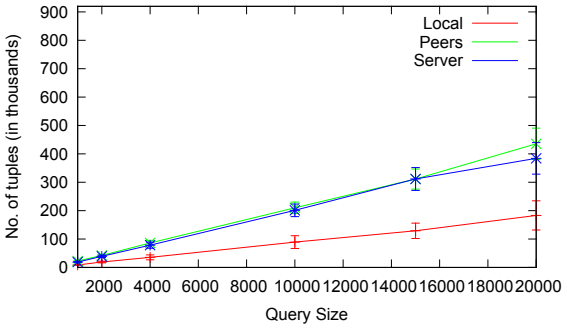

Figure 6.23: Query Size: Tuples' Origin

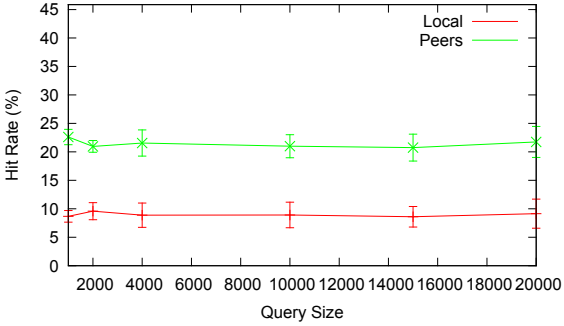

Figure 6.24: Query Size: Hit Rate

crease with the number of clients since more semantic regions are indexed in the distributed index.

6.1.7 Distributed Index

The next experiment evaluates the general system peformance when the parameters of the distributed index are varied. The minimum level (f_{min}) is set to 10, while the maximum level (f_{max}) varies from 10 to 24.

Figures 6.31 and 6.32 illustrate the results of this experiments. The duration of the query rewriting process first begins to decrease because, while increasing the maximum level, caches entries and indexed into an increasing number of distributed quad structures, and thus, the distributed index becomes more efficient. A further increase of the this

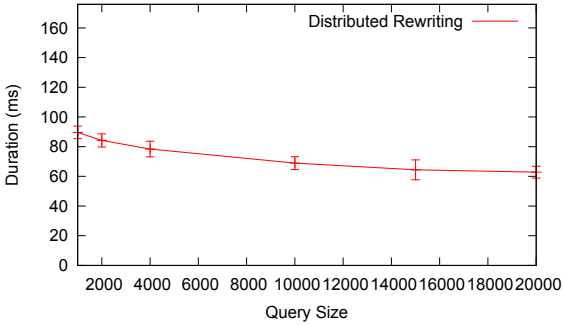

Figure 6.25: Query Size: Rewriting Duration

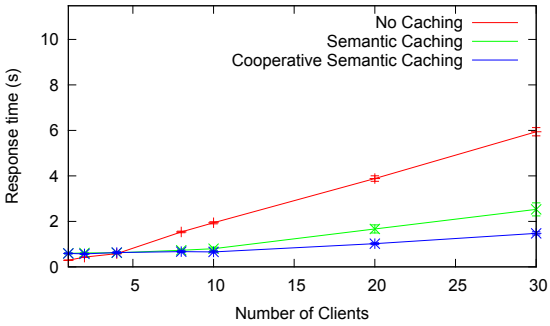

Figure 6.26: Number of Clients: Response Time

maximum level decreases the performance due to the communication overhead of having many quad structures. A similar behavior is exhibited when looking and the response time (Figure 6.32).

6.1.8 Updates

The last experiment investigates how update statements influence the performance of the two caching approaches. The size of clients' cache is 64 MB. The workload consists of a sequence of alternative selection and update sessions. Selection sessions are generated similarly with the first experiment. Update sessions contain a number of updates statements which modify a single tuple chosen randomly based on the normal dis-

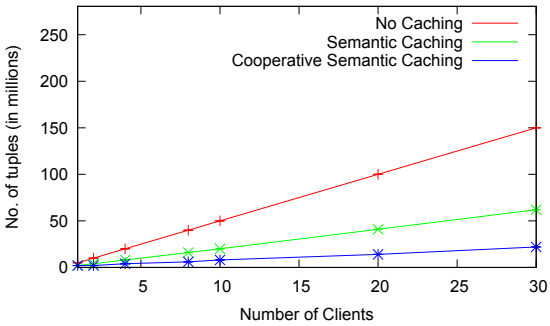

Figure 6.27: Number of Clients: Server Tuples

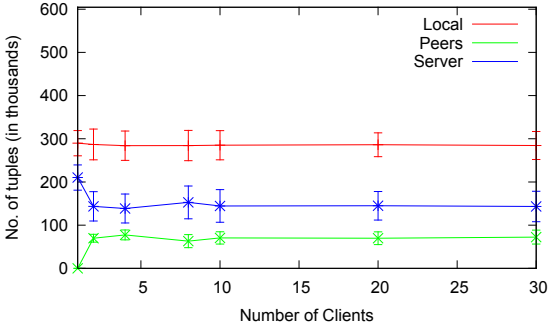

Figure 6.28: Number of Clients: Tuples' Origin

tribution used for the selection sessions. The number of update statements per session is varied from 0 to 150.

While the number of update statements per session increases, the performance of both caching systems starts to decrease because update statements invalidate an increasing number of cache entries. Thus, both the query response time (Figure 6.33) and the number of tuples sent by database server (Figure 6.34) increase. For the cooperative caching approach, increasing the number of update statements causes an increase of tuples that originate from the database server and decreases the amount of data returned from either the local or remote caches (Figure

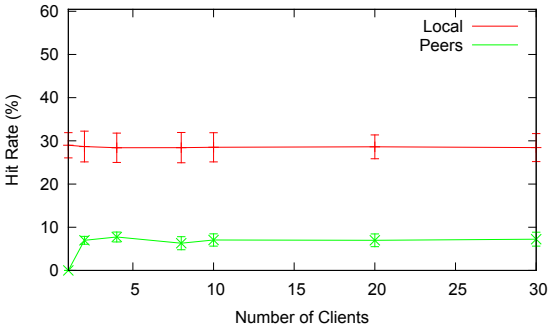

Figure 6.29: Number of Clients: Hit Rate

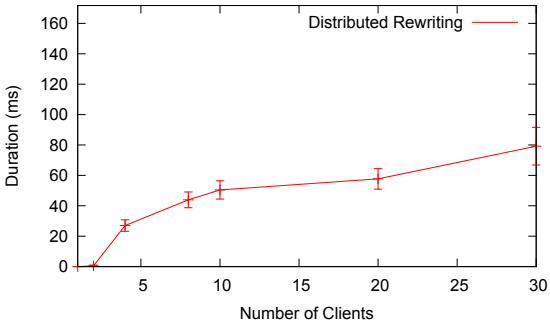

Figure 6.30: Number of Clients: Rewriting Duration

6.35). Both the local and peers hit rate decrease with the increase of update statements due to the invalidation of cache entries (Figure 6.36).

A further evaluation was performed to determine how the update level (f_{update}) influences the performance of the system. The update level is varied from 4 to 18. These values are consistent with the size of selections and, thus, with the sizes of the interval convered by semantic regions. The size of an update session is set to 50, which is a reasonable value for read-intensive workloads.

For lower update levels, the overhead of the distributed rewriting increases because the number of timestamps transferred grows (Figure 6.38). This also causes an increase of the response time (Fig-

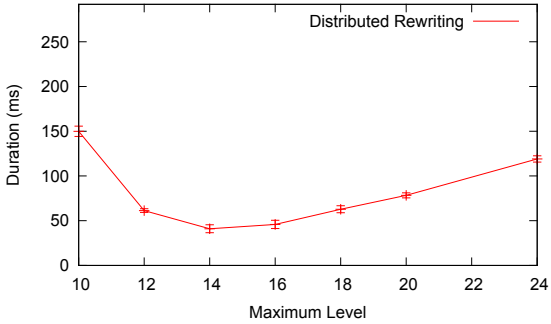

Figure 6.31: Distributed Index: Rewriting Duration

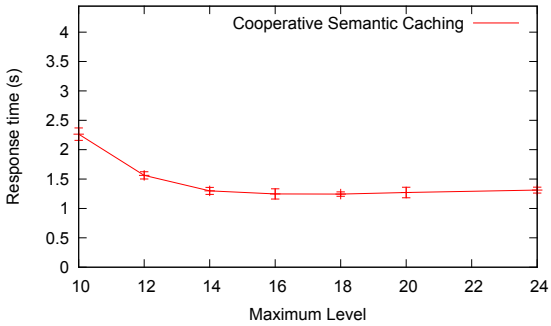

Figure 6.32: Distributed Index: Response Time

ure 6.37). For large update levels, the overhead of the distributed rewriting decreases but also the number of semantic regions which are invalided by update statements increases, and thus the performance of the caching system decreases.

6.2 Use Case 1: CoopSC within Cloud Environments

Similarly with the general CoopSC evaluation, the two cloud scenarios, introduced in Chapter 4.5, were evaluated using the Wisconsin benchmark [14] relation. The workloads consists of single attributes selections on *unique1* attribute, generated as described in the previous sec-

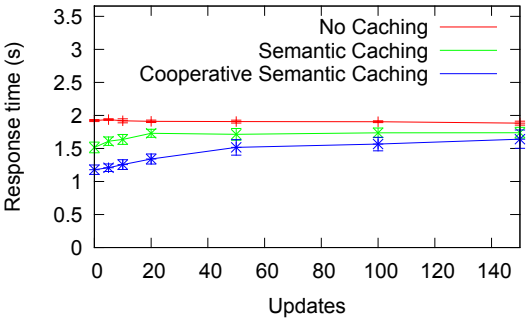

Figure 6.33: Updates: Response Time

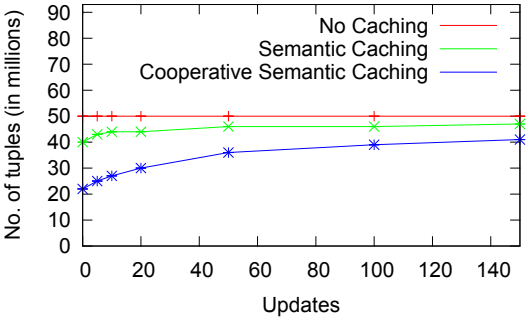

Figure 6.34: Updates: Server Tuples

tions. These experiments look both at performance and also of the economic impact of CoopSC. Thus, based on the charging schemes, the amount of money paid for data transferred are also calculated In each experiment, three measurements are made: query response time (a), amount of data sent by database server and amount of money paid for data transfer (b).

6.2.1 Scenario A

In this experiment, database is located in Zurich, while clients are running in nodes provided by Rackspace [76] cloud infrastructure. The size of clients' caches are varied from 0 to 192 MB. This experiment uses 5

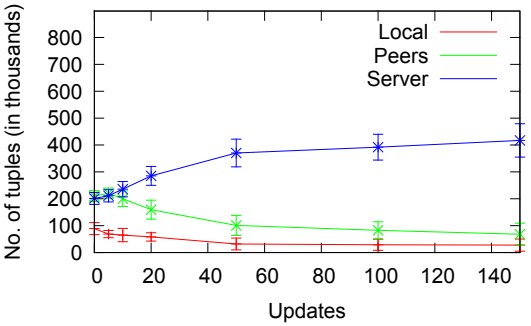

Figure 6.35: Updates: Tuples' Origin

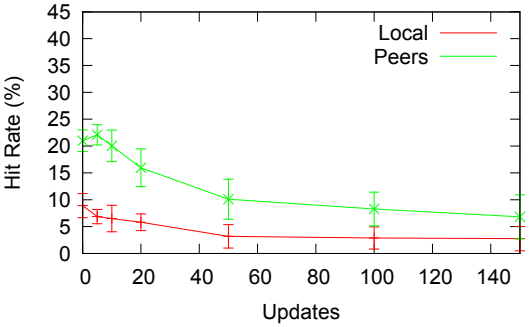

Figure 6.36: Updates: Hit Rate

clients. The workloads are generated in a similar matter with the general system evaluation.

The results of this experiment are presented in Figures 6.39 and 6.40. Due to the instability of the resource provided to virtual machines by Rackspace response time measurements (Figure 6.39) show a high degree of instability and thus, the performance-wise benefits of CoopSC are not evident. Figure 6.40 shows the amount of data sent by database server during experiments. Taking Rackspace's charging scheme into consideration, the amount of money that has to be paid for data transferred is computed and illustrated in the figure. Thus, the economic-

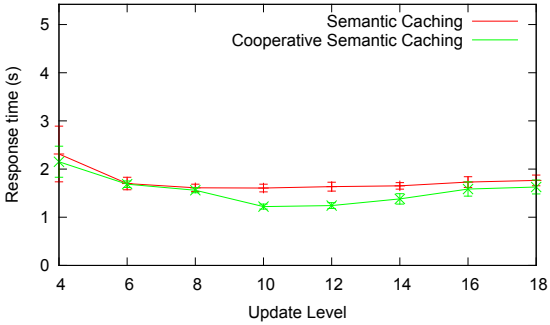

Figure 6.37: Update Level: Response Time

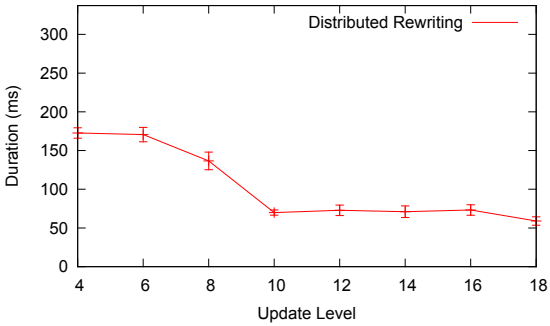

Figure 6.38: Update Level: Rewriting Duration

wise benefits of using a cooperative caching solution in this scenario are shown.

6.2.2 Scenario B

In this scenario the database server runs in a large Amazon EC2 [10] instance while clients run in nodes located in the EMANICSLab testing environment. Two experiments were performed: the first experiment measures how cache size influences the performance of system, while the second experiment varies the update rate. The cost of data transfer is computed using Amazon EC2's pricing scheme.

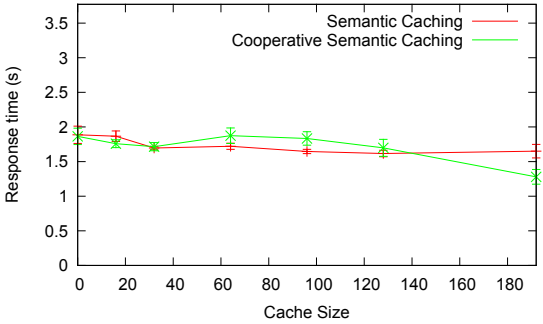

Figure 6.39: Cloud Scenario A: Response Time

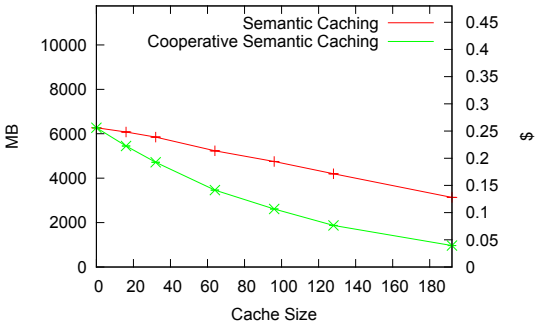

Figure 6.40: Cloud Scenario A: Server Tuples

The workload of the first experiment is generated similarly with Scenario A. Analyzing response time (Figure 6.41), for small cache sizes, the difference between the two approaches is reduced, because hit rates are small in both scenarios and database server has to handle executions of most queries. While the cache sizes increase, the benefits of the cooperative caching approach become more visible. In the semantic caching approach, the amount of data sent by database server is reduced, because database server only sends parts of queries which are missing from local cache. The cooperative approach further decreases this amount of data because clients can also transfer tuples from caches of other peers. Re-

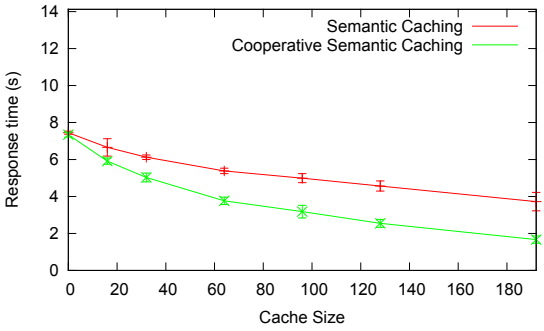

Figure 6.41: Cloud Scenario B: Response Time

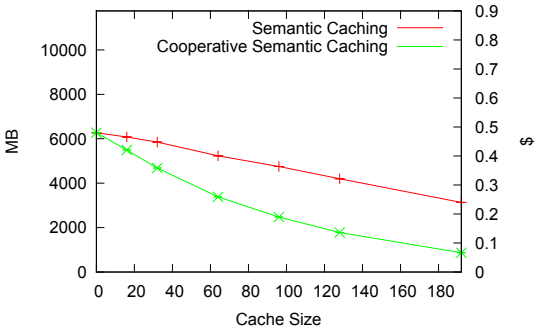

Figure 6.42: Cloud Scenario B: Server Tuples

ducing the amount of data also determine the reduction in the cost of data transfer which is clearly visible in Figure 6.42).

The second experiment investigates how update statements influence the performance of the cooperative caching approach. The size of clients' cache is 64 MB. The workload consists of a sequence of alternative selection and update sessions. Selection session are generated similarly with the first experiment. Update sessions contain a number of updates statements which modify a single tuple chosen randomly based on the normal distribution used for selection sessions. The number of update statements per session is varied from 0 to 150. Figure 6.43 and 6.44 illustrates the results of this experiment. While the number of update statements per session increases, the performance of the caching system starts to decrease because update statements invalidate an in-

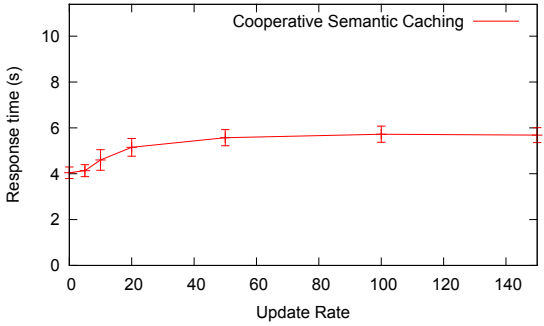

Figure 6.43: Cloud Scenario B (Updates): Response Time

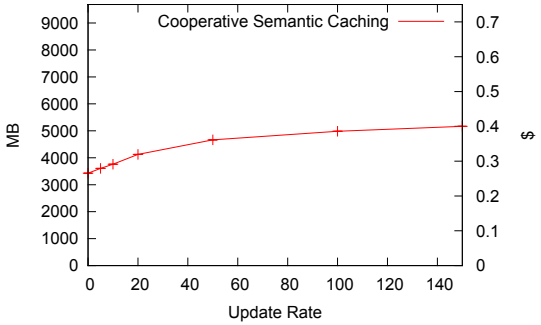

Figure 6.44: Cloud Scenario B (Updates): Server Tuples

creasing number of cache entries. Thus, both query response time and number of tuples sent by database server increase. The cost of data transfer follows the same trend.

6.3 Use Case 2: CoopSC-based Network Traffic Analysis

The NMCoopSC architectures (introduced in Chapter 4.6) was implemented and evaluated using a PostgreSQL database server and a number of clients that execute, in parallel, single indexed attribute selection queries on *starttime* attribute. The EMANICSLab [39] research testing network was used for this evaluation. Clients are running in 10 nodes located across Europe, while the database server runs on a more powerful machine located in Zurich. During the evaluation, three scenarios were

used: cooperative semantic caching approach, classic semantic caching, and no caching approach.

The evaluation was done using a relation of about 60 million real-life flow records, which were collected during an interval of 50 hours from two operational nodes running inside the PlanetLab [74] network. Each query is a range selection on the *starttime* attribute (Example: select * from flows where 1253190752 < starttime and starttime < 1253230752). It is assumed that analyzers' access patterns have a semantic locality. This means that each analyzer has an interest area in which most queries are executed. The width of this interest area is also an important testing variable. In order to express the locality of queries, the access pattern of different analyzers is modeled using the normal distribution. Its standard deviation determines the width of interest area. Thus, for each client, the center-points of queries were randomly chosen to follow a normal distribution curve with different standard deviations. For each experiment, clients first execute warm-up queries until cache is filled. The response time, for each client, is calculated by averaging the response time of 8 testing sessions of 20 queries each. The error bar is calculated using these 8 values. These values were chosen in order to increase the precision of the measurements. For each scenario, the total amount of data sent by the database server is also measured. Furthermore, for the cooperative caching scenario, in each session, numbers of tuples that originated from the local cache, remote caches, and the database server are determined. *Local* and *peers* hit rates are computed for the cooperative caching scenario. The local hit rate is defined as the percent of tuples from result sets that originated from the local cache, while the peers hit rate refers to tuples that were returned from caches of other clients.

Thus, in each experiment, four measurements are made: the first two compare the cooperative semantic caching approach with the classic semantic caching and no caching scenario, in relation to (a) query response time and (b) amount of data sent by database server. The other two measurements refer only to the cooperative caching approach and (c) determine tuples' origin and (d) hit rates.

The distributed index was configured with the following parameters $f_{min} = 5, f_{max} = 9$. These values determine a good distribution of cache entries within the distributed index.

6.3.1 Cache Size

The first experiment measures how the variation of the size of clients' caches influences the performance of the two caching approaches for single-attribute selections. The size of clients' caches are varied from 0 to 192 MB. These cache sizes were chosen it order to make the overall behavior of the system visible when the current workload is used. This experiment uses 10 clients. Each query returns records which were generated in an interval of 300 seconds (about 10,000 tuples). These workloads have standard deviations of 15,000. The means of the Gaussian curves are distributed uniformly over the range of the *starttime* attribute. The difference between the means of two consecutive clients is 9,000. This difference makes sure that interest areas of different clients are well separated and the queries asks for time intervals which are stored in the database.

Analyzing the response time (Figure 6.45a) for small cache sizes, the difference between the two approaches is reduced, because hit rates are small in both scenarios and the database server has to handle executions of most queries. While these cache sizes increase, the benefits of the cooperative caching approach become more visible. In the semantic caching approach, the number of tuples sent by the database server is reduced (Figure 6.46), because the database server only sends parts of queries which are missing from the local cache. The cooperative approach further decreases the number of tuples sent, because clients can also transfer tuples from caches of other peers. Figure 6.47 illustrates the origin of tuples for the cooperative caching approach. For small cache sizes, most tuples are returned from the database server. As cache size increases, both the number of tuples returned from the local cache and caches of other client increase. Hit rates have a similar behavior (Fig-

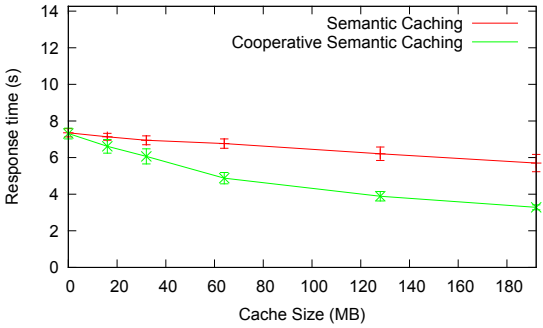

Figure 6.45: NMCoopSC Cache Size: Response Time

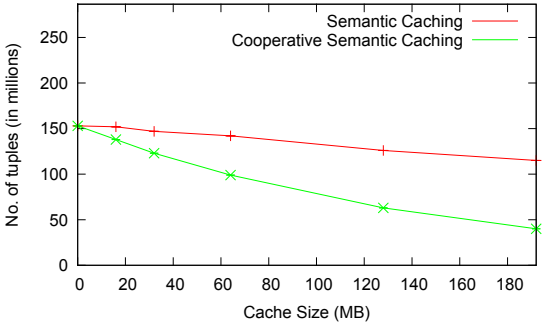

Figure 6.46: NMCoopSC Cache Size: Server Tuples

ure 6.48). For small cache sizes both local and peers hit rate are reduced. Increasing the cache size causes the increase of both hit rates.

6.3.2 Locality

The second experiment measures how varying queries' locality influences the performance of both caching approaches. The size of the clients' cache is 64 MB. The experiment uses 10 clients. Workloads' standard deviations are varied from 5,000 to 50,000 seconds.

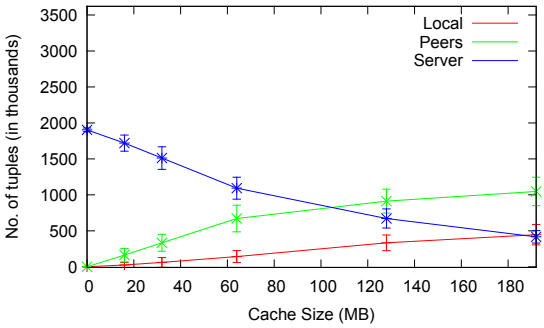

Figure 6.47: NMCoopSC Cache Size: Tuples' Origin

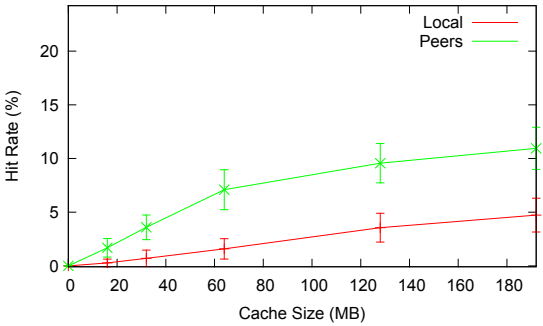

Figure 6.48: NMCoopSC Cache Size: Hit Rate

Figures 6.49, 6.50, 6.51, and 6.52 illustrate the results of this experiment. Increasing standard deviation of workloads decreases access locality and thus, performance of both caching systems decreases (Figures 6.49 and 6.50). In the cooperative caching approach, increasing the standard deviation causes number of tuples returned from the local cache to decrease, while the number of tuples returned from the server increases. The number of tuples returned from remote peers initially increases, because lowering access locality increases the need to cooperate. A further increase of standard deviation decreases also the amount of data returned from remote peers because with a general low access locality, relevant entries will not be found in other peers (Figure 6.51).

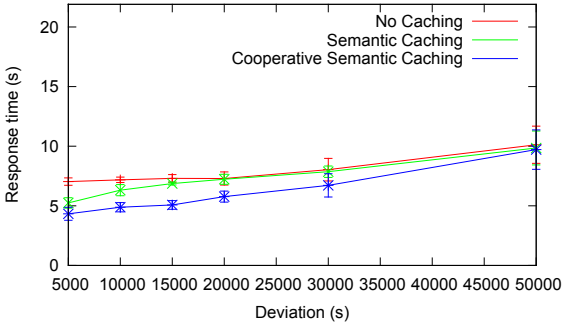

Figure 6.49: NMCoopSC Deviation: Response Time

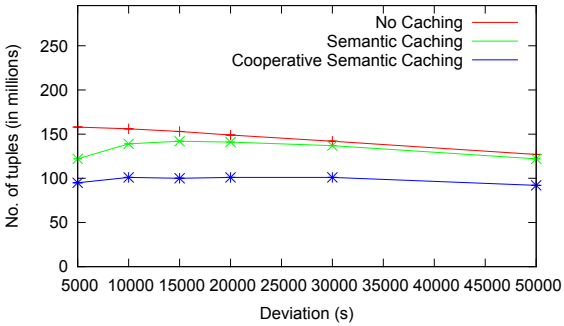

Figure 6.50: NMCoopSC Deviation: Server Tuples

Local hit rate decreases with the increase of the standard deviation while peers hit rate initially increases and then decreases (Figure 6.52).

6.3.3 Query Size

The next experiment measures how the size of selections influences the performance. The size of selections is varied from 50 to 600 seconds. Key results of this experiment are presented in Figures 6.53, 6.54, 6.55 and 6.56. As the size of selections increases, the response times (Figure 6.53) and the amount of data sent by database server (Figure 6.54) also increase. Similarly to the first experiment, the semantic caching ap-

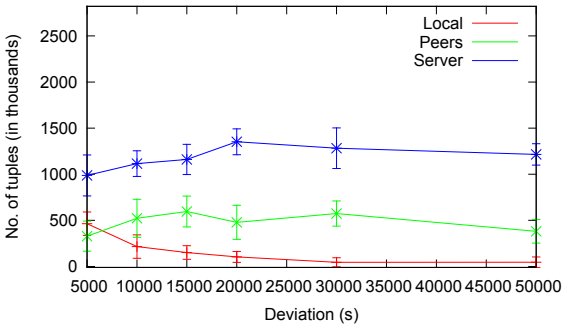

Figure 6.51: NMCoopSC Deviation: Tuples' Origin

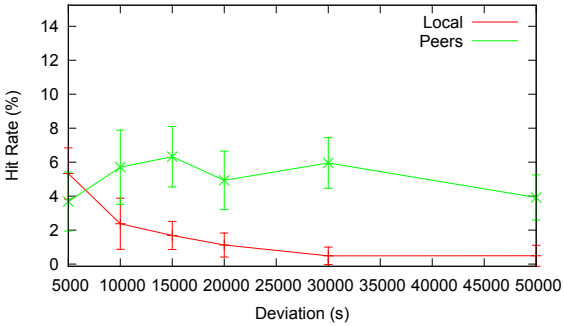

Figure 6.52: NMCoopSC Deviation: Hit Rate

proach is more efficient than the no-caching approach because database server only handles parts of queries which can not be answered using the cache. The cooperative caching solution outperforms the local caching approach due to the increase of hit rate of the cache system. The hit rate remains constant because the same proportion of queries are answered using cache (Figure 6.56).

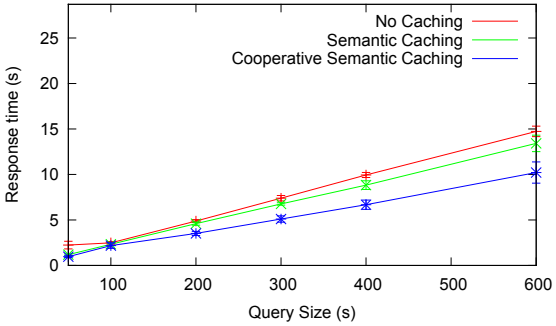

Figure 6.53: NMCoopSC Query Size: Response Time

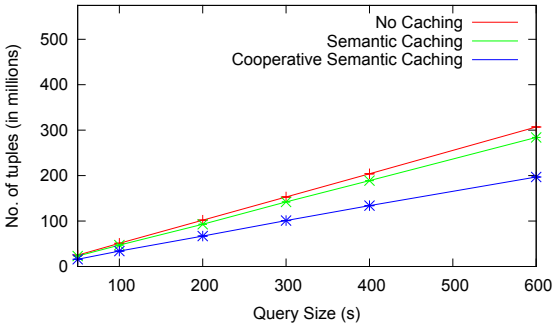

Figure 6.54: NMCoopSC Query Size: Server Tuples

6.4 Chapter Summary

The evaluation, described in this chapter, shows that using CoopSC within a distributed client-server database environment improves the performance of range selection queries. Experiments were performed for measuring average query response time, amount of tuples sent by database server, hit rates and tuples' origins. The overhead of the CoopSC approach was also shown by measuring the duration of the distributed rewriting.

Update statements were also evaluated and results indicate that CoopSC has beneficial effects when read-intensive workload are executed. Choosing a appropriate value for the update level emerged as an important factor that has to be considered.

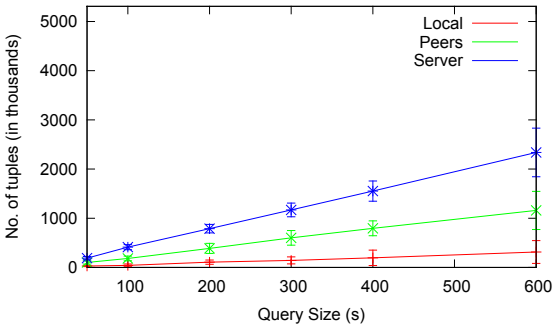

Figure 6.55: NMCoopSC Query Size: Tuples' Origin

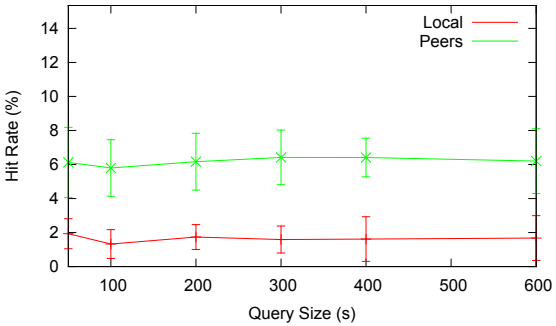

Figure 6.56: NMCoopSC Query Size: Hit Rate

The economic and performance-wise benefits of using CoopSC within real-life cloud environment were shown by running experiments inside Rackspace [76] and and Amazon EC2 [11] cloud infrastructures.

Finally, the NMCoopSC architecture was evaluated using real-life NetFlow data and results show that a cooperative database semantic caching approach improves the performance of off-line NetFlow-based network monitoring approaches.

7

Summary and Conclusions

The CoopSC approach determines a cooperative semantic caching architecture, that optimizes the execution of database queries by caching old query results in order to answer new queries, allowing clients to share their cache entries in a cooperative matter. CoopSC supports n-dimensional range select-project queries. Update queries are also handled. The design of the CoopSC approach was described and major details outlined. The proposed approach was evaluated and compared with the classic semantic caching approach. These evaluation results show that CoopSC, especially by applying distributed principles and the P2P overlay techniques in particular, reduces the response time of range selection queries and the amount of data sent by database server for read-intensive workloads. The benefits for workloads with a significant number of updates statements are limited due to the increased invalidation of cache entries.

Based on the general CoopSC approach, two cloud based scenarios were determined and evaluated inside real-life cloud environments, outlining the main performance and economic benefits. Furthermore, the CoopSC approach was used in designing the NMCoopSC architecture,

which improves the handling of NetFlow records within NetFlow-based network monitoring solutions.

7.1 Conclusions per Research Problem

In what follows, based on the content of this thesis, the six main research questions which were introduced in Section 1.2 are restated and answered in a summarized way.

A. **How can cooperative aspects be integrated in the classic semantic caching approach?**

As described in Chapter 3, the CoopSC approach integrates cooperative aspects into the semantic caching approach by allowing clients to share their local cache entries. Each query is split into *probes* (parts of query which are answered using local cache), *remote probes* (sub-queries executed using remote caches) and *remainders* (which are executed on database server). These sub-queries are determined during the *query rewriting* process. Cache entries are indexed in a distributed data structure built on top of a P2P overlay which is formed by all clients which are interrogating a specific database server. This distributed index in based on the MX-CIF quad tree data structure.

B. **How to handle update statements in the context of the cooperative semantic caching approach?**

Update statements are handled with a cooperation from the database server (c.f., Section 3.4). The approach uses the same quad space division as the distributed index. A specific *update level* is chosen by the administrator. *Virtual timestamps* are stored for each quad from the specified update level. A trigger mechanism increments these timestamps when modifications are performed. Semantic regions store the values of these timestamps. Before executing a query, current snapshots are determined by finding which quads intersect given query and their corresponding virtual timestamps. The query rewriting process ignores and discards old cache entries.

C. **Does the cooperative caching approach improve the performance of database solutions?**

In order to determine the performance of the cooperative semantic caching approach, a prototype CoopSC system was fully implemented. The system was evaluated (c.f., Chapter 6) and compared with the classic semantic approach and a no-caching scenario. The results of this extensive evaluation clearly show that the cooperative caching approach improves both query response time and amount of data sent by database servers.

D. **How does the update handling mechanism impact the performance of the cooperative caching approach?**

The results of the evaluation (c.f., Section 6.1.8) show the CoopSC approach behaves well when read-intensive workload are executed. The general performance decrease while the rate of update increases. Also, choosing the right value for the *update level* is important in regards to the performance.

E. **How can the cooperative caching approach be applied within cloud-computing database solutions and NetFlow-based traffic monitoring applications and which are the benefits?**

Two practical scenarios were determined for applying the CoopSC approach within cloud environments (c.f., Chapter 4.5). Also, the NMCoopSC architecture (c.f., Chapter 4.6) was developed in order to improve the handling of NetFlow records in the context of NetFlow based traffic monitoring solutions.

The evaluation of the two cloud scenarios clearly show the economic benefits of applying the cooperative semantic caching within real-life cloud environments. The performance-wise benefits are not clearly visible in all experiments due to the instability of cloud providers (c.f., Section 6.2).

Furthermore, the NMCoopSC architecture was evaluated (c.f., Section 6.3) and the results of these evaluations show that both response

time and amount of transferred data are improved when applying a cooperative semantic caching approach.

7.2 Future Work

A potential future work direction consists in determining a cache replacement policy which optimizes the global performance of clients in the cooperative semantic caching approach, rather than the local individual performance. In the CoopSC approach, each clients manages its local cache independently by apply the LRU (Least Recently Used) replacement policy. Thus, a popular cache entry can be present in the caches of many clients. This might have negative performance consequences since the global cache size is limited in size. The general performance of the system might be further optimized if a global cache replacement policy is applied which limits the number of times a particular entry is stored.

The CoopSC approach optimized the performance of queries in the context of relational database management systems. Further investigations can be performed in order to determine if the approach can be adapted and applied in non-relational database management systems, such as *key-value storage solutions* (e.g., Apache Cassandra [5], Project Voldemort [92]) or *document-oriented database systems* (e.g., Apache CouchDB [6], mongoDB [68]) which, nowadays, have become popular since they are suited for many applications which do not require strict ACID transactional support. Since, usually, these systems use a semi-structured type system, adapting and using CoopSC efficiently might presented additional challenges [84].

Further investigations can also be conducted in determining new scenario for applying CoopSC within cloud-based environments. For example, a potential new scenario might consist in running a database server and a number of CoopSC-enabled nodes inside the cloud environment while clients (which do not need to cache query results) run outside. Client requests are redirected to the CoopSC-enabled nodes which can reduce the load of the database server. The number of such

CoopSC node can be elasticity increased or decreased depending on the current load of the system. Determining the performance benefits of such scenarios is a potential future work direction.

7.3 Final Conclusion

This thesis studied how cooperative aspects can be integrated into the classic semantic caching solution. In order to determine the feasibility of such an approach, a CoopSC prototype system was designed, implemented and evaluated. The general approach was also applied in the context of cloud-based database solutions and for improving the performance of NetFlow-based network monitoring solutions.

Therefore, this thesis shows that adding a cooperative dimension to the classic semantic caching solution is technical feasible and such an approach improves the general system performance for a wide variety of real-life use cases in which multiple nodes use range queries for interrogating database servers. Geographical Information Systems (GIS) are a good potential use case since two-dimensional range interrogations are commonly used by many such solutions.

Netflow-based network monitoring solutions are another good potential use case of the CoopSC approach which was investigated by this thesis. The experiments which were performed during the evaluation of this use-case (c.f., Section 6.3) show that the CoopSC approach improves the performance of offline network analyzers which access Netflow-based data stored in a logically centralized storage solution. Moreover, the load of such a storage solution is also reduced. Thus, CoopSC can make the execution of tasks such as charging, accounting or intrusion detection more efficient.

Another important result of this thesis is the design and the implementation of the update handling mechanism. This mechanism extends the set of potential use cases of the cooperative semantic caching approach also to applications in which modifications are performed in the database. While the update mechanism is flexible enough to support all types of workloads, the system evaluation shows that, from a perfor-

mance perspective, only read-intensive workloads benefit from using the CoopSC approach. For write-intensive workloads the performance benefits do not always exist because many cache entries are invalidated.

Finally, this thesis shows that using a cooperative semantic caching approach presents also economic advantages when applied within cloud computing environments, because most cloud providers charge, in a pay-as-you-go matter, the amount of data transferred between the cloud and the outside world. Hence, cloud-based software solutions can benefit from using CoopSC from both a technical and an economic perspective.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34. Baltimore, Maryland, USA, June 1995.

[2] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: paving the way for an adaptive database cache. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 23–34, Berlin, Germany, September 2003.

[3] Amazon rds. http://aws.amazon.com/rds, June 2012.

[4] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *Proceedings of 19th International Conference on Data Engineering (ICDE)*, pages 821–831, Bangalore, India, March 2003.

[5] Apache cassandra. http://cassandra.apache.org, June 2012.

[6] Apache couchdb. http://couchdb.apache.org, June 2012.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley, USA, February 2009.

[8] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Building an Object-Oriented Database System*, pages 1–20. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1992.

[9] Franz Aurenhammer. Voronoi diagrams — a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, September 1991.

[10] Amazon.com web services: Products and services. http://aws.amazon.com/product, June 2012.

[11] Amazon.com ec2: Ec2 frequently answered question. http://aws.amazon.com/ec2/faqs, June 2012.

[12] Francois Bancilhon, Gilles Barbedette, Véronique Benzaken, Claude Delobel, Sophie Gamerman, Christophe Lécluse, Patrick Pfeffer, Philippe Richard, and Fernando Velez. The design and implementation of o2. In *Lecture Notes in Computer Science on Advances in Object-Oriented Database Systems*, pages 1–22. Springer-Verlag Inc., New York, New York, USA, 1988.

[13] Ingmar Baumgart and Sebastian Mies. S/kademlia: A practicable approach towards secure key-based routing. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–8, Hsinchu, Taiwan, December 2007.

[14] Dina Bitton and Carolyn Turbyfill. A retrospective on the wisconsin benchmark. In *Readings in Database Systems (2nd ed.)*, pages 422–441. Morgan Kaufmann Publishers Inc., 1988.

[15] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Martin May, and Anukool Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM*, pages 159–164, Rio de Janeiro, Brazil, October 2006.

[16] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 532–543, Santa Barbara, California, USA, May 2001.

[17] Michael J. Carey, David J. DeWitt, Goetz Graefe, David M. Haight, Joel E. Richardson, Daniel T. Schuh, Eugene J. Shekita, and Scott L. Vandenberg. The exodus extensible dbms project: an

overview. In *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1990.

[18] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 357–366, Denver, Colorado, USA, May 1991.

[19] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 264–276, Boston, Massachusetts, USA, June 2006.

[20] Li Chen, Elke A. Rundensteiner, and Song Wang. Xcache: a semantic caching system for xml queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 618–618, Madison, Wisconsin, June 2002.

[21] Boris Chidlovskii and Uwe M. Borghoff. Semantic caching of web queries. *The VLDB Journal*, 9(1):2–17, March 2000.

[22] Chimera. http://p2p.cs.ucsb.edu/chimera, June 2012.

[23] Cisco netflow. http://www.cisco.com/web/go/netflow, June 2012.

[24] KKimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Application of sampling methodologies to network traffic characterization. In *Proceedings of the 3th ACM SIGCOMM*, pages 194–203, San Francisco, California, USA., September 1993.

[25] Benoit Claise. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information. http://tools.ietf.org/html/rfc5101, January 2008.

[26] Benoit Claise, Andrew Johnson, and Juergen Quittek. Packet sampling (psamp) protocol specifications. http://tools.ietf.org/html/rfc5476, March 2009.

[27] William G. Cochran. *Sampling Techniques*. Wiley, 1987.

[28] Nicholas Coleman, Rajesh Raman, Miron Livny, and Marvin Solomon. A peer-to-peer database server based on bittorrent. Technical Report 10891, School of Computing Science, Newcastle University, 2008.

[29] George Copeland and David Maier. Making smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–325, Boston, Massachusetts, June 1984.

[30] Michael D. Dahlin, Olph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 267–280, Monterey, California, USA, November 1994.

[31] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, pages 330–341, Bombay, India, September 1996.

[32] Yves Denneulin, Cyril Labbé, Laurent d'Orazio, and Claudia Roncancio. Merging file systems and data bases to fit the grid. In *Proceedings of 3rd International Conference on Data Management in Grid and P2P Systems*, pages 13–25, Bilbao, Spain, August 2010.

[33] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, Seattle, Washington, United States, June 1998.

[34] David J. DeWitt, Philippe Futtersack, David Maier, and Fernando Vélez. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pages 107–121, Brisbane, Queensland, Australia, August 1990.

[35] Laurent d'Orazio and Mamadou Kaba Traoré. Semantic caching for pervasive grids. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 227–233, Cetraro, Calabria, Italy, September 2009.

[36] Nick Duffield, Derek Chiou, Benoit Claise, Aaron Greenberg, Matthias Grossglauser, and Jennifer Rexford. A framework for packet selection and reporting. http://tools.ietf.org/html/rfc5474, March 2009.

[37] Nick Duffield, Carsten Lund, and Mikkel Thorup. Learn more, sample less: Control of volume and variance in network measurement. *IEEE Transactions on Information Theory*, 51(5):1756–1775, December 2005.

[38] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[39] Emanicslab. http://www.emanicslab.org, June 2012.

[40] Flow-tools. http://www.splintered.netlsw/flowtools, June 2012.

[41] Michael Franklin and Michael J. Carey. Client-server caching revisited. In *Proceedings of the International Workshop on Distributed Object Management (IWDOM)*, pages 493–503, Edmonton, Alberta, Canada, August 1992.

[42] Michael J. Franklin and Michael J. Carey. Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, 22(3):315–363, September 1997.

[43] Gogrid website: Gogrid cloud services. http://www.gogrid.com, June 2012.

[44] Jonathan Goldstein and Perake Larson. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Record*, 30(2):331–342, June 2001.

[45] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[46] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, pages 332–347, Jerusalem, Israel, January 1999.

[47] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: problems, techniques, and applications. In *Materialized Views*, pages 145–157. MIT Press, Cambridge, Massachusetts, USA, 1999.

[48] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, Massachusetts, USA, June 1988.

[49] Alon Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, December 2001.

[50] Takahiro Hara. Cooperative caching by mobile clients in push-based information systems. In *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM)*, pages 186–193, McLean, Virginia, USA, November 2002.

[51] Christina Hoffa, Gaurang Mehta, Tim Freeman, Ewa Deelman, Kate Keahey, Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *Proceedings of the 4th IEEE International Conference on eScience*, pages 640–645, Indianapolis, India, USA, December 2008.

[52] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Information Systems (TOIS)*, 5(1):70–95, January 1987.

[53] Cay S. Horstmann. *Practical Object-Oriented Development in C++ and Java*. John Wiley & Sons, Inc., 1997.

[54] Haibo Hu, Jianliang Xu, Wing Sing Wong, Baihua Zheng, Dik Lun Lee, and Wang-Chien Lee. Proactive caching for spatial queries in mobile environments. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 403–414, Tokyo, Japan, April 2005.

[55] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*, pages 213–222, Monterey, California, July 2002.

[56] Björn Þór Jónsson, María Arinbjarnar, Bjarnsteinn Þórsson, Michael J. Franklin, and Divesh Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3):302–331, August 2006.

[57] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, , and Kian-Lee Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 25–36, Madison, Wisconsin, June 2002.

[58] Björn Karlsson. *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005.

[59] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(1):35–47, January 1996.

[60] Won Kim, Jorge F. Garza, Nat Ballou, and Darrell Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, March 1990.

[61] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34(10):50–63, October 1991.

[62] Alon Y. Levy, Alberto O. Mendelzon, and Yehoshua Sagiv. Answering queries using views (extended abstract). In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 95–104, San Jose, California, USA, May 1995.

[63] Kostas Lillis and Evaggelia Pitoura. Cooperative xpath caching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 327–338, Vancouver, Canada, June 2008.

[64] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, , and Hui Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM*, pages 165–176, Pisa, Italy, September 2006.

[65] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[66] Microsoft sql azure. http://www.microsoft.com/en-us/sqlazure/default.aspx, June 2012.

[67] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[68] mongodb. http://www.mongodb.org, June 2012.

[69] Cristian Morariu, Peter Racz, and Burkhard Stiller. Script: A framework for scalable real-time ip flow record analysis. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 278–285, Osaka, Japan, April 2010.

[70] Cristian Morariu and Burkhard Stiller. Distributed architecture for real-time traffic analysis. In *Proceedings of 4th International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, pages 171–174, Zurich, Switzerland, June 2010.

[71] Nfdump. http://nfdump.sourceforge.net, June 2012.

[72] Bill Nickless. Combining cisco netflow exports with relational database technology for usage statistics, intrusion detection, and network forensics. In *Proceedings of the 14th USENIX Conference on System Administration*, pages 285–290, Boston, Massachusetts, USA, June 2000.

[73] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Proceeding of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 178–190, Cambridge, Massachusetts, USA, March 2002.

[74] Planetlab. http://www.planet-lab.org, June 2012.

[75] Wolfgang Pree. *Design patterns for object-oriented software development*. Reading, Mass.: Addison-Wesley, 1994.

[76] Rackspacecloud website: Rackspacecloud service. http://www.rackspacecloud.com, June 2012.

[77] George Reese. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2000.

[78] Qun Ren and Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 210–221, Boston, Massachusetts, USA, August 2000.

[79] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, California, USA, June 1995.

[80] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[81] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[82] Arie Segev and Jooseok Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[83] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer*

communications (SIGCOMM), pages 149–160, San Diego, California, United States, August 2001. ACM.

[84] Michael Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53(4):10–11, April 2010.

[85] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2000.

[86] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16(2):165–178, April 2007.

[87] Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The gmap: a versatile tool for physical data independence. *The VLDB Journal*, 5(2):101–118, April 1996.

[88] Andrei Vancea, Laurent d'Orazio, and Burkhard Stiller. A scalable cooperative semantic caching (coopsc) approach to improve range queries. In *Proceeding of the 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 45–54, Orlando, Florida, USA, October 2011.

[89] Andrei Vancea, Guilherme Sperb Machado, Laurent d'Orazio, and Burkhard Stiller. Cooperative database caching within cloud environments. In *Proceedings of 6th International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, pages 14–25, Luxembourg, Luxembourg, June 2012.

[90] Andrei Vancea and Burkhard Stiller. Answering queries using cooperative semantic caching. In *Proceedings of 3rd International Conference on Autonomous Infrastructure, Management and Security (AIMS)*, pages 203–206, Enschede, The Netherlands, June 2009.

[91] Andrei Vancea and Burkhard Stiller. Coopsc: A cooperative database caching architecture. In *Proceedings of the 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*, pages 223–228, Larissa, Greece, June 2010.

[92] Project voldemort. http://project-voldemort.com, June 2012.

[93] Yongdong Wang and Lawrence A. Rowe. Cache consistency and concurrency control in a client/server dbms architecture. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–376, Denver, Colorado, USA, May 1991.

[94] W. Kevin Wilkinson and Marie-Anne Neimat. Maintaining consistency of client-cached data. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, pages 122–133, Brisbane, Queensland, Australia, August 1990.

[95] Liangzhong Yin and Guohong Cao. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(1):77–89, January 2006.

[96] Baihua Zheng, Wang-Chien Lee, and Dik Lun Lee. On semantic caching and query scheduling for mobile nearest-neighbor search. *Wireless Networks - Special issue: Pervasive Computing and Communications*, 10(6):653–664, November 2004.

[97] Tanja Zseby, Maurizio Molina, Nick Duffield, Saverio Niccolini, and Federico Raspall. Sampling and filtering techniques for ip packet selection. http://tools.ietf.org/html/rfc5475, March 2009.

Listing of Figures

Listing of Tables

Other Author Publications

7.4 Papers

- Fabio V. Hecht, Patrick Poullie, Andrei Vancea, Burkhard Stiller, Attacks on Internet Names. Readme Alumni - Das Bulletin der Alumni Wirtschaftsinformatik Universität Zürich, Alumni Witschaftsinformatik Universität Zürich, No. 28, pages 14-15, October 2012.
- Christos Tsiaras, Martin Waldburger, Guilherme Sperb Machado, Andrei Vancea, Burkhard Stiller, The Design of a Single Funding Point Charging Architecture. EUNICE 2012, Budapest, Hungary, pages 1-12, August 2012.
- Andrei Vancea, Guilherme Sperb Machado, Laurent d'Orazio, Burkhard Stiller, Cooperative Database Caching within Cloud Environments. Dependable Networks and Services, 6th International Conference on Autonomous Infrastructure, Management and Security (AIMS 2012), June 2012.
- Andrei Vancea, Laurent d'Orazio, Burkhard Stiller, Optimization of Flow Record Handling by Applying a Decentralized Cooperative Semantic Caching Approach. 13th IEEE/IFIP Network Operations and Management Symposium (NOMS) , April 2012.
- Andrei Vancea, Laurent d'Orazio, Burkhard Stiller, A Scalable Cooperative Semantic Caching (CoopSC) Approach to Improve Range Queries. CollaborateCom 2011: 7th International Conference on Collaborative Computing: Networking, Applications and Worksharing, October 2011.
- Martin Waldburger, Burkhard Stiller, Guilherme Sperb Machado, Andrei Vancea, AMAAIS - Accounting and Monitoring of AAI Services. AMAAIS - A4-Mesh Meeting, pages 1-5, February 2011.
- Martin Waldburger, Burkhard Stiller, Guilherme Sperb Machado, Andrei Vancea, AMAAIS - Accounting and Monitoring of AAI Services. AAA/SWITCH Info Day, pages 1-8, January 2011.

- Andrei Vancea, Burkhard Stiller, CoopSC: A Cooperative Database Caching Architecture. 19th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2010), IEEE, July 2010.
- Burkhard Stiller, Fabio Hecht, Maurizio Lo Bosco, Peter Racz, Guilherme Sperb Machado, Andrei Vancea (Eds.), Internet Economics V . IFI Technical Report, July 2010.
- Andrei Vancea, Burkhard Stiller, A Cooperative Semantic Caching Architecture for Answering Selection Queries. IFI Technical Report, University of Zurich, September 2009
- Andrei Vancea, Burkhard Stiller, Answering Queries Using Cooperative Semantic Caching. 3nd International Conference on Autonomous Infrastructure, Management and Security (AIMS 2009), Springer, July 2009.
- Andrei Vancea, Michael Grossniklaus and Moira C. Norrie, Database-Driven Web Mashups, International Conference on Web Engineering (ICWE 2008), Yorktown Heights, New York, USA, July 2008
- Moira C. Norrie, Michael Grossniklaus, Corsin Decurtins, Alexandre de Spindler, Andrei Vancea and Stefania Leone, Semantic Data Management for db40, In Proceedings of ICOODB 2008, 1st International Conference on Object Databases, Berlin, Germany, March 2008
- Andrei Vancea, Michael Grossniklaus and Moira C. Norrie, Generic Proxies - Supporting Data Integration Inside the Database, In Proceedings of DOA 2007, 9th International Symposium on Distributed Objects and Applications (Poster), Vilamoura, Algarve, Portugal, November 2007
- Sergiu Nedevschi, Dan Olinic, Iulian Popa, Florin Rusu, Andrei Vancea and Calin Homorodean, DICOM Compliant Structured Reporting Environment, BIOSIGNAL 2004, Brno, Czech Republic
- Liviu Miclea, Szilard Enyedi, Lucian Busoniu, Ioan Stoian and Andrei Vancea, Limited Resource Systems Testing with Microagent Societies, European Test Symposyum 2004, Ajaccio, France
- Dan Olinic, Calin Homorodean, Sergiu Nedevschi, Florin Rusu, Alexandru Smeu, Iulian Popa and Andrei Vancea, A Proposal for Structured Diagnosis Reporting in Echocardiography Using a DICOM Compliant Environment, Computers in Cardiology, Chicago, 2004
- Sergiu Nedevschi, Dan Olinic, Iulian Popa, Florin Rusu, Alexandru Smeu, Andrei Vancea and Calin Homorodean, Tools and methodology

for DICOM SR in echocardiography, AQTR 2004, Cluj-Napoca, Romania
- Alin Suciu, Kalman Pusztai and Andrei Vancea, Prolog Server Pages, Roedunet International Conference 2003 Iasi, Romania

7.5 Technical Reports

- Burkhard Stiller, Károly Farkas, Fabio Hecht, Guilherme Sperb Machado, Andrei Vancea, Martin Waldburger (Eds.), Communication Systems V. IFI Technical Report, IFI 2012.06, pages 1-89, August 2012.
- Burkhard Stiller, Károly Farkas, Fabio Hecht, Guilherme Sperb Machado, Patrick Poullie, Flavio Santos, Christos Tsiaras, Andrei Vancea, Marin Waldburger, Internet Economics VI. IFI Technical Report, No. IFI-2012.02, pages 1-147, April 2012.
- Burkhard Stiller, Hasan,Fabio Hecht, Guilherme Sperb Machado, Andrei Vancea, Martin Waldburger (Eds.), Mobile Systems V. IFI Technical Report, No. IFI-2011.05, pages 1-100, November 2011.
- Burkhard Stiller, Hasan, Fabio Hecht, Guilherme Machado, Andrei Vancea, Martin Waldburger, Communication Systems IV . IFI Technical Report, pages 1-81, January 2011.
- Burkhard Stiller, Fabio Hecht, Maurizio Lo Bosco, Peter Racz, Guilherme Sperb Machado, Andrei Vancea (Eds.), Internet Economics V . IFI Technical Report, July 2010.
- Andrei Vancea, Burkhard Stiller, A Cooperative Semantic Caching Architecture for Answering Selection Queries. IFI Technical Report, University of Zurich, September 2009.
- Burkhard Stiller, Thomas Bocek, Fabio Hecht, Cristian Morariu, Peter Racz, Andrei Vancea, Martin Waldburger (Eds.), Communication Systems III. IFI Technical Report, June 2009.

Acknowledgments

I would first like to thank my advisor Burkhard Stiller for giving me the opportunity to work in the Communication Systems Research Group and for his support throughout my research and writing work. I offer my sincere appreciation for his guidance during the completion of my doctoral thesis. In addition, I would also like to thank Torsten Braun for his insightful feedback on my thesis.

I would like to express my deep appreciation to all my colleagues (Cristian, Peter, Fabio, Martin, Thomas, Andri, Guilherme, Christos, Patrick) which provided a wonderful environment for doing research. I will miss our interesting discussions during coffee breaks.

Many thanks to my friends (Horațiu, Tudor, Laurențiu, Bogdan, Alex) and to my family for providing me with moral support and encouragement throughout the last four years.

Curriculum Vitae

Andrei Vancea was born in Cluj-Napoca, Romania, on September 22, 1980. He received his Master Degree from Technical University of Cluj-Napoca, Romania, in June 2004. While holding an ERASMUS scholarship he developed his Master Thesis at the Swiss Federal Institute of Technology (ETHZ). For his master thesis he developed a cooperative graphical editor.

Starting from December 2005 until December 2008, he was a junior research assistant at ETH Zurich, Global Information Systems Group, Switzerland. In January 2009 he joined the University of Zurich, Communication Systems Group, where he worked until December 2012, as PhD Student, supervised by Prof. Dr. Burkhard Stiller. His areas of expertise include peer-to-peer systems, database management systems and caching architectures.